Operational Profiling of OS Drivers

Vom Fachbereich Informatik der Technischen Universität Darmstadt
genehmigte

Dissertation

zur Erlangung des akademischen Grades eines Doktor-Ingenieur (Dr.-Ing.)
vorgelegt von

Constantin Sârbu

aus Bukarest, Rumänien

Referenten:
Prof. Neeraj Suri, Ph.D.
Prof. Christof Fetzer, Ph.D.

Datum der Einreichung: 06. März 2009
Datum der mündlichen Prüfung: 25. Mai 2009

Darmstadt 2009
D17

Summary

Operating Systems (OS's) constitute the operational core for computing devices. In order to facilitate their applicability to a variety of hardware platforms, OS's have evolved into complex componentized software entities whose key function is to provide applications access to the system resources. Fundamentally, the provided system services inherently depend on the stability of the underlying OS. Within the OS, the key components that dominate the cause of OS failures are the *device drivers* (DDs), precisely the OS parts designed to enhance the OS's support for hardware. Despite intensive efforts to elevate the DDs' robustness level by employing varied test paradigms, the existing testing approaches still exhibit very high failure rates. Hence, the central premise behind this thesis involves the characterization of the DD's operational profile, and using it for focusing subsequent testing to the functionality areas likely to be exercised over the DD deployment.

This thesis develops two novel and distinct methodologies to capture and analyze the operational profile of DDs. The first – termed as the *operational profile* (OP) – is based on the characterization of the I/O traffic between a selected DD and the rest of the OS kernel. The second – termed *execution path profile* (EPP) – observes the functional calls made by the respective DD in the operational phase, thus revealing the code paths followed at runtime. Both presented approaches are directly applicable to DD binaries as they do not require source-code level access to any of the involved OS components.

This thesis develops the concepts and methodology for effectively profiling the operational behavior of DDs. First, a state model is introduced for describing a DD and its complete state space. Experimentally, we show that the DD's *operational state space* (OSS) – the subset of states visited at runtime – represents only a small fraction of the total state space, thus highlighting the areas to be tested. Subsequently, occurrence- and duration-based quantifiers are defined for each of the DD states belonging to the OSS. This enables test prioritization and workload comparisons which are the key factors for testing. This conceptual process's effectiveness is tested using extensive case studies including over fifty Windows XP and Vista DDs.

The developed EPP is complementary to the OP as it discovers execution hotspots as frequently traversed DD code paths. To highlight the execution hotspots, a DD monitoring and code path analysis methodology is presented and tested using actual Windows DD's. Code paths are identified as call sequences to kernel functions implemented externally to the selected DD. String similarity metrics are used to compute the relative similarity among the inferred code paths. Based on likeness, the code paths are grouped into equivalence classes helping identify execution hotspots. These hotspots constitute primary targets for testing.

Overall the thesis develops novel profiling approaches for testing generalized OS's. The research is also validated on actual Windows XP and Vista OSs.

Kurzfassung

Betriebssysteme (BS) bilden den operativen Kern eines jeden Computers. Für eine einfache Anwendbarkeit auf verschiedenen Hardware-Plattformen haben sich BS zu komplexen, aus Komponenten bestehenden Software-Einheiten entwickelt, deren Hauptaufgabe darin besteht Anwendungen Zugriff auf Systemressourcen zu ermöglichen. Grundsätzlich hängen die bereitgestellten Systemdienste inhärent von der Stabilität des unterliegenden BSs ab. Die Komponenten, die innerhalb eines BSs die dominierenden Verursacher von Ausfällen des BSs sind, sind die Gerätetreiber, genauer gesagt die Teile des BSs, die entwickelt wurden um die Hardware-Unterstützung des BSs zu verbessern. Trotz intensiver Anstrengungen das Robustheitsniveau von Gerätetreibern durch Anwendung verschiedener Testparadigmen zu heben, zeigen existierende Testmethoden nach wie vor äußerst hohe Fehlerraten. Die Charakterisierung des operativen Profils von Gerätetreibern und dessen Anwendung während der Entwicklung von Gerätetreibern zum zielgerichteten Testen von mit hoher Wahrscheinlichkeit aufgerufenen Funktionen bildet den Kernbeitrag der vorliegenden Arbeit.

Die vorliegende Arbeit beschreibt zwei neue Methoden um ein operatives Profil von Gerätetreibern zu erstellen und zu analysieren. Die erste Methode — *operational profile* (OP) genannt — basiert auf der Charakterisierung der Ein-/Ausgabe-Kommunikation zwischen einem ausgewählten Gerätetreiber und dem Rest des BS-Kerns. Die zweite Methode — *execution path profile* (EPP) genannt — verfolgt die vom entsprechenden Gerätetreiber während der Betriebsphase gemachten Funktionsaufrufe, wobei die zur Laufzeit durchschrittenen Pfade im Code aufgezeigt werden. Beide vorgestellten Ansätze können direkt auf die ausführbaren Dateien von Gerätetreibern angewendet werden, weil kein Zugriff auf den Quellcode einer beteiligten BS-Komponente benötigt wird.

Die vorliegende Arbeit präsentiert Konzepte und Methoden zur effektiven Profilierung des operativen Verhaltens von Gerätetreibern. Zunächst wird ein Zustandsmodell eingeführt um einen Gerätetreiber und den kompletten Zustandsraum zu beschreiben. Wir zeigen mit Hilfe von Experimenten, dass der operative Zustandsraum (die Teilmenge von Zuständen, die zur Laufzeit besucht werden) eines Gerätetreibers nur einen kleinen Anteil des gesamten Zustandsraums ausmacht und somit die Gebiete im Zustandsraum hervorhebt, die vornehmlich durch Tests abgedeckt werden müssen. Anschließend werden für jeden Zustand des Gerätetreibers Quantifizierer, basierend auf dem (zeitlichen) Auftreten des Zustandes, definiert. Dies ermöglicht Priorisierung von Tests und den Vergleich von Arbeitslasten, welche die Schlüsselfaktoren für genaues und angemessenes Testen sind. Die generelle Effektivität des gesamten Prozesses wird anhand von ausführlichen Fallstudien, die mehr als fünfzig Windows XP und Vista Gerätetreiber umfassen, erprobt.

Das EPP, welches ebenso in dieser Arbeit vorgestellt wird, ist komplementär zum OP weil es Ausführungs-Hotspots in Form von häufig durchschrittenen Pfaden im Code auffindet. Um solche Ausführungs-Hotspots hervorzuheben wird

eine Methode zum Monitoring von Gerätetreibern und zur Code-Pfad-Analyse präsentiert und an echten Windows-Gerätetreibern getestet. Code-Pfade werden als Sequenzen von BS-Kern-Funktionsaufrufen charakterisiert, die außerhalb des gewählten Gerätetreibers implementiert sind. String-Ähnlichkeitsmetriken werden herangezogen um die relative Ähnlichkeit zwischen den abgeleiteten Code-Pfaden zu bestimmen. Die Code-Pfade werden, basierend auf Wahrscheinlichkeiten, in Äquivalenzklassen unterteilt um das Identifizieren von Ausführungs-Hotspots zu erleichtern. Solche Hotspots bilden primäre Ziele für das Testen.

Zusammenfassend entwickelt die vorliegende Arbeit neue Profilierungsansätze zum Testen allgemeiner BS. Die Forschungsarbeit wird an den realen Betriebssystemen Windows XP und Windows Vista validiert.

Acknowledgements

Almost six years have passed since I started the work that eventually crystallized into this PhD thesis. Looking back from today's perspective, I must admit that there was a lot of hard work but fun was always present. It was hard as I had to learn how to do research, how to present my work and how to be critical with other people's work when it came to writing conference reviews. It has been fun when socializing with the members of our group or celebrating birthdays over home-made, international-flavor dishes or cakes.

There are many people that helped me to whom I am very thankful. First I'd like to thank my advisor for teaching me everything I know about doing research. From him I learned that there is always a higher gear, but also that research can be (sometimes) fun. Thanks, *Neeraj*!

Then, I am very thankful to all past and present DEEDS group's members for helping me, listening to my boring presentations and giving me feedback on early drafts of my papers. Hoping not to forget anyone of the past and present guys, many thanks to *Andréas, Adina, Robert, Ripon, Brahim, Peter, Dan, Marco, Faisal, Matthias, Majid, Azad, Piotr* and *Vinay*. Also, special thanks go to *Birgit, Sabine* and *Ute* for helping me with various paperwork, for correcting and improving my German and all other circumstances related to living in Germany. A great many thanks also to Prof. Christof Fetzer for accepting to be my co-advisor.

I'd also like to thank my parents and brother for their continual support and love which spanned the geographical distance separating us in all these years. Thank you, *Mama, Tata* and *Mai*!

I saved the best for the last: Thank you *Adina* for all the understanding, love and support that you always found for me all these years! Thank you *Daniel* for making everything so easy for me!

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Problem Context

Why are device drivers an important source of OS-related failures despite sustained test efforts and what can be improved?

The diversity of computational entities and services is proliferating. Examples range from tiny embedded systems acting as nodes in wireless sensor networks and ubiquitous mobile phones and PCs to powerful server clusters used by Internet companies such as Google, Amazon or Ebay with each utilizing specific software (SW) applications.

As it is not viable to develop new *operating systems* (OSs) for each entity, a componentized approach is increasingly used with *commercial-off-the-shelf* (COTS) OSs chosen as key building blocks of such systems, and acting as mediators between the hardware parts and the installed SW applications. Such OSs are transparent to the users and their main function is to provide the services enabling the SW applications to perform their tasks in a correct, timely and reliable manner.

The OS interface to the hardware is represented by *device drivers* (DDs). They are often implemented as add-on components to the OS kernel, and are responsible for handling the I/O operations with the hardware. With hundreds of devices that can be attached to each ordinary computing system (about 250 DDs in a regular Windows XP or Vista installation [Mendonca and Neves, 2007]), the DD code represents a significant share of the total OS code. For example, in Linux about 70% of the total lines of kernel code belongs to DDs [Swift et al., 2005].

Unfortunately, the large size of DD code combined with the rapid feature-

driven development cycles often implies limited DD testing coverage. Consequently, DDs are released still containing undetected defects, thus leading to overall OS outages in the operational phase. The generalized OS failure is caused by the direct and un-restricted interaction of faulty DDs with critical OS kernel structures. These observations are confirmed by results of the OS reliability research community, from multiple independent [Ganapathi et al., 2006], academic [Albinet et al., 2004; Arlat et al., 2002; Chou et al., 2001; Durães and Madeira, 2003; Swift et al., 2005] and industry [Murphy et al., 2006; Simpson, 2003] results.

While multiple, sophisticated test approaches (applicable in various stages of a DD's development life-cycle) exist, the community continues to witness OS failures generated by insufficiently or inadequately tested DDs. Most testing approaches (formal, statistical or random-based) aim at exploring the full DD operational state space. The immense state space size naturally precludes complete testing to be viable due to testing time and cost issues. Hence, parts of the DD often remain un-tested by design, and defects inherently arise post-shipment as the DD's broad functionality is executed.

Our approach to improve DD testing is not to target the generic full state space but to identify specific subsets of DD code that are actually executed in operational mode of the DD, i.e., DD *operational profiles*. Hence, these "*actual*" operational modes constitute areas to focus testing on.

Consequently, this thesis addresses the problem of highlighting the operational DD states by monitoring the communication interfaces of the black-box level DDs within the OS kernel. Using the I/O traffic obtained through monitoring, an *operational profile* of the targeted driver is built and is subsequently used to ascertain the subset of the total DD state space which requires focused testing. While being completely non-intrusive, the presented methods help the selection of relevant test cases and define the actual test space that needs to be covered. Existing test campaigns benefit from our DD operational profiles by adopting a prioritized coverage of the operational states, enabling faster discovery of the defects likely to occur in the field.

This chapter presents the general DD testing problem context alongside with a discussion of its causes. The main ideas driving the research in this thesis are also introduced here and refined as a set of conceptual and experimental research questions. A brief summary of the thesis contributions is also presented in this chapter.

1.1 Operational Profiling: A Prerequisite for Accurate and Adequate Testing

This section introduces the context of the OS robustness problem and discusses why DDs continue to constitute an important cause of OS failures despite significant advancements in OS testing research. Subsequently, we explain why current DD testing paradigms are inadequate. Our central ideas for improving DD testing via operational phase profiling are discussed, together with defining the constraints the solution must observe in order to ensure general applicability.

1.1.1 OS Complexity – Between Necessity and Burden

Initially, computing systems were designed for dedicated tasks with tightly controlled resource management. For instance, in *batch systems* the main computation task and the I/O functions were physically decoupled (running on separate machines). Due to the fact that powerful machines were expensive, a *mainframe* (responsible for actual computation) was seconded by several, inexpensive machines (responsible for reading punched cards, writing onto magnetic tapes and printing). Such systems did not have a holistic OS; the task of organizing the jobs and feeding the cards and tapes was the responsibility of human operators of the batch system.

The later development of *multiprogramming* marked the appearance of OSs as we now know them. Multiprogramming allows computers to load multiple jobs into memory, permitting a job to use the CPU while other jobs are waiting for I/O to complete. The CPU utilization was thus enhanced while minimizing the required operator intervention. This was the primary job of a simple program loaded onto the computer, called *operating system*. The OS's task was to manage the attached I/O devices and the jobs stored in memory, and to decide which one to run next when the current job blocks for I/O.

The continual evolution of hardware formed a dominant catalyst for OS development. The appearance of faster CPUs, larger and faster storage (available at cheaper costs), fostered the establishment of the OS role as a mediator between the user and the available hardware resources. The development of new input devices (i.e., mice), new user interfaces (i.e., GUIs) paralleled the advent of novel, innovative OS concepts and mechanisms like *multi-tasking*, *multi-processing* or later, *multi-threading*. Thus, the OSs slowly became transparent to the users, who could therefore concentrate on system applications instead of directly being aware of the existence of the OS and

hardware characteristics of the used machine. The more automation offered by the OS, the more overhead was offloaded from the user of the computing system. For reference, many well established books like [Tanenbaum, 2001] or [Silberschatz et al., 2004] discuss in depth the OS history and multiple other OS-related aspects.

Unfortunately, the continual addition of new features and support for new peripherals also introduced more defects in the code associated with the OS, proving to be a two-edged sword in terms of computer reliability. The program code associated with the OS itself increased tremendously, current OSs containing tens of millions of lines of code. For instance, [Amor et al., 2005] summarizes the work presented in [Wheeler, 2000, 2001] and lists the total lines of source code in millions (MSLOC) for several Linux distributions (see Table 1.1). As the presented research reflects the situation in 2005 for the listed Linux distributions, it is interesting to note the authors' observation that "*Debian total size (both in MSLOC and in number of packages) is doubling every two-three years*", this observation highlighting the actual trend.

Table 1.1: The evolution of GNU/Linux OSs in terms of lines of code (adapted from Amor et al. [2005]; 1 MSLOC = 10^6 lines of code).

Distribution Name	**Release date**	**MSLOC**
Red Hat 5.2	April 1998	12
Red Hat 6.0	April 1999	15
Red Hat 6.2	March 2000	17
Red Hat 7.1	April 2001	30
Red Hat 8.0	September 2002	50
Red Hat 9.0	March 2003	53
Debian 2.0	July 1998	25
Debian 2.1	March 1999	37
Debian 2.2	August 2000	59
Debian 3.0	July 2002	105
Debian 3.1	June 2005 (est.)	229
Fedora Core 2	May 2004	67
Fedora Core 4 (pre.)	May 2005	76

Other popular COTS OSs have comparable sizes in terms of lines of code and also follow the same trend towards a significant size increase from version to version. For instance, the Apple's OS X v10.4 (code-named "Tiger", also an UNIX-based OS) released in April 2005 is evaluated at 86 MSLOC [Jobs, 2006]. Summarizing the data from various sources such as [Maraia, 2005]

and [Hiner, 2008], Table 1.2 lists approximate values for different versions of Microsoft Windows.

Table 1.2: The evolution of Microsoft Windows OSs in terms of lines of code (adapted from Hiner [2008]; Maraia [2005]; 1 MSLOC = 10^6 lines of code).

Product (Release Name)	**Release date**	**MSLOC**
NT 1.0 (Windows 3.1)	July 1993	4–5
NT 2.0 (Windows 3.5)	September 1994	7–8
NT 3.0 (Windows 3.51)	May 1995	9–10
NT 4.0 (Windows 4.0)	July 1996	11–12
NT 5.0 (Windows 2000)	December 1999	29+
NT 5.1 (Windows XP)	October 2001	35–40
NT 5.2 (Server 2003)	April 2003	50
NT 6.0 (Windows Vista)	January 2007	50+

Logically, *code testing* was increasingly used to avoid shipping code containing errors despite the increasing size and complexity of SW. In his book, Glenford Myers defines SW testing as "*the process of executing a program with the intent of finding errors*" [Myers, 2004]. According to this definition, to systematically find the errors present in a program a tester should execute the program in all possible ways, that is, cover all possible execution paths. While this task might seem trivial when using proper testing tools, reaching 100% coverage even for very small programs is difficult (if not impossible [Kaner et al., 1999]).

Myers provides an illustrative example for the effect of program complexity on testing [Myers, 2004, chap. 2, pp. 11–12]. The "*10- to 20-statement program*" used as an example has approximately 10^{14} possible execution paths, generated by few loops and branches in the program code. To cover all of them, a testing strategy that needs only one second to test each path would finish in 3.2 million years, an obviously unrealistic task.

Given the immense size and inherent complexity of the code base of actual OSs, and starting from the premise that 100% coverage is idealistic for testing it in reasonable time, OS testers customarily resort to various abstractions to reduce the test space they need to cover. Such abstractions (i.e., decisions on which program parts need to be rigourously tested and which not) are usually based on tester's personal expertise and *rules-of-thumb*. Also, in order to remove the human variable from this equation, (semi-)automatic test paradigms which test at random were developed. While proving their applicability and effectiveness to various test scenarios, such practices often leave large parts of the program un-tested, leading to program releases containing

defects likely to be reached only in the operational phase of the respective program.

Moreover, a situation specific to COTS OSs is the limited amount of resources for testing. As in many commercial products, OS projects often have to limit themselves to the available budgets, size and experience of personnel (developers) and deadlines. Under such circumstances, sometimes OS components have to be released on the market even before the test cycle initially planned is finished, irrespective of the reached coverage level.

Beside the effects of the OS size and complexity on testing, a related aspect is its *robustness*, defined as "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" [IEEE, 1990]. In order to provide the specified level of service to the users of a computing system equipped with an OS even in the presence of perturbations, modern OSs have to cope with the effects of failures generated by various entities. Among these entities which are potentially faulty and the OS has to mask or withstand their failures are user applications, OS components, hardware components and even inexperienced users. In the following, we attempt to explain why device drivers are still faulty despite sustained test efforts.

1.1.2 Faulty Drivers Despite Testing?

Currently, the largest (and also the most evolving) code part of an OS is its interface to the hardware devices, as represented by the *device drivers* (DD).

For a better understanding of the concepts further presented in this thesis, a brief description of the functioning of a DD is overviewed. Figure 1.1 illustrates the role of the DD in the mechanism for performing I/O, as typically implemented by most of the current COTS OSs. An I/O request is issued by an user application to the kernel of the OS. Specialized structures of the OS kernel receive the request, and forward it to the responsible DD (i.e., a "`file read`" request is forwarded to the hard-drive DD, a "`print file`" request to the printer DD, and so forth). The respective DD performs the necessary activity on the associated device located into the hardware space (actual file reading or actual printing).

As soon as the I/O activity finishes, the DD forwards the results (content of the read file or print acknowledgement) to the application that issued the original I/O request, via the OS structures responsible for preparing the I/O results in the format expected by the application.

The DDs play an important role on the I/O path acting as mediators between the OS and the various hardware devices, thus their correct, timely and reliable provisioning of results is crucial to the user programs.

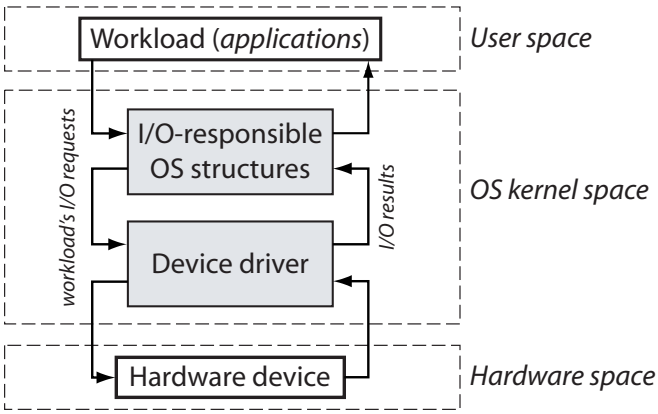

Figure 1.1: The role of the device drivers in the I/O path.

Unfortunately, recent research [Albinet et al., 2004; Arlat et al., 2002; Durães and Madeira, 2003; Ganapathi et al., 2006; Simpson, 2003; Swift et al., 2005] has shown that DDs constitute a dominant cause of OS failures. In the following we summarize what we believe to be the main causes explaining why DDs tend to contain more defects than the OS base:

DD code immaturity: Currently, OS kernels have reached a certain maturity not to be the primary failing component *per se*. The main reason is that major changes to kernel functionality occur fairly seldomly and they are incremental – a new version of an OS often being an extension of the previous one. The functionalities proved to perform well in the past version are usually kept. Instead, nowadays many new DDs are produced everyday (as 3rd party offerings) to support the ever-increasing volume of hardware peripherals, leading to an immense pool of immature DDs on the market. The error reporting facility of Windows Vista enabled Microsoft's researchers to estimate the number of different devices attached to Vista systems to 390000, while the DD population is fulminant, increasing every day with 25 new and 100 revised DDs on average [Orgovan, 2008].

Inconsistent OS interfacing: As most DDs are built by 3rd party vendors or by hardware producers and not by the developers of the OS, the communication interfaces of the DD with the OS represent a key design attribute, unfortunately also the source of failure. This fact can be explained by the total reliance of the DD developers on the interface specifications which are not always complete and correct, leading the DDs (or the whole OS) to unspecified states [Oney, 2003; Orwick and Smith, 2007]

Human factor: DDs are inherently diverse and complex due to their add-on nature, and consequently are difficult to develop using a standard architectural basis. Usually, DD development requires teamwork, and each developer's experience, style, etc. differs from others'. Hence, different developers produce code of different quality. This results in DDs whose parts are heterogenous from the viewpoint of the amount of defects they contain; also, overall some DDs are more defective than others [Möller and Paulish, 1993].

Under a feature-driven market pressure, DDs are often released without exhaustive testing, usually exhibiting a higher defect density compared to the more mature OS kernel [Chou et al., 2001]. In addition, the sample of drivers used for testing is often different from deployed systems. Consequently, the coverage obtained from the limited test configurations does not entirely match the wider spectrum of deployed system configurations.

In addition, the existing testing techniques are often ineffective at revealing all the defects that a DD contains. In an attempt to understand this effect, we list below the main reasons we believe that explain the inefficacy of DD testing.

DD complexity: The DDs installed in a common OS setup account for a large fraction of code commonly associated with the OS – about 70% in Linux – leading to tens of millions of lines of kernel code. A direct effect of this is that the testing techniques must resort to a selection of the sub-parts of a DD to be tested, as covering 100% of it is unattainable.

DD location: Most of the modern COTS OSs are implemented as monolithic kernels, and faulty DDs are usually installed in kernel space, where they freely interact with critical OS structures. Hence, a failure of a DD may lead to a generalized OS failure. General SW testing tools are inefficient for DDs due to the limited accessibility inside the OS kernel space, thus warranting special testing approaches.

Black-box approach: DDs are typically delivered as binaries constraining potential testing campaigns to black-box strategies. While several testing approaches able to test a program using only its binary image (without requiring source code access) exist, their application to kernel DDs is limited as the binaries of the DDs are kept in the kernel memory area where the access of user-level programs is restricted.

Code coverage: Despite the efforts to cover as much of the DD code as possible while observing the limitations imposed by various test constraints (method, resources, costs, etc.) some DD parts are left out by

the test process. Unfortunately, these parts are often responsible for generating most of the failures observed in the field, after DD release.

Test case selection: The process of building and ultimately sampling the set of appropriate test cases to be applied to the DD from the set of all available test cases is based on tester's experience or it is often done at random, by automatic tools [Whittaker, 2000, 2003]. As the efficiency of a test campaign is measured by the ratio between found errors and test costs, such a practice does not always yield optimal results, important resources are wasted as a result of the unfortunate choice of test cases.

While our research addresses many facets of the mentioned DD test inefficacy arguments, the latter two (*code coverage* and *test case selection*) represent the central focus of the research presented in this thesis, as key aspects to improving actual DD test paradigms.

1.1.3 Core Thesis Idea: Driver State Space Profiling

On this basis an important consideration is the DD testing under "*field*" conditions. While multiple sophisticated static testing techniques for DDs exist [Ball et al., 2004; Mendonca and Neves, 2007; Nagappan et al., 2005], the choice of a relevant workload is key to exercising a DD in its actual operational domain. Although the *operational profile* of a DD is difficult to capture and later to reproduce for testing, once obtained it can bring significant advantages over static testing techniques by identifying the functionalities actually executed, their sequence and occurrence patterns. Using this valuable information, we believe that subsequent test campaigns can primarily target code likely to be executed in the field, therefore decreasing the time required to find the defects with high operational impact. Consequently, developers and system integrators are required to envision workloads that realistically mimic the manner in which the DD (or the whole system) will be used [Weyuker, 1998], i.e., the DD's operational profile. The more accurate the profile, the more effective a test campaign can be developed to test the DD state space.

To better illustrate these aspects, Figure 1.2 qualitatively depicts the *total state space* of a DD and also its *operational space* visited by the DD while executing under the effect of a workload. The operational state space is obviously a subset of the total state space. As the DD is not fault-free, the state space contains some parts that are defective. As long as the defective states are not reached in the field (i.e., do not belong to the operational

state subset), the DD performs according to its specifications. In contrast, if a faulty area is exercised at runtime, the obtained operational space will include it. From this perspective, to ensure a fault-free execution of the DD at runtime, the purpose of testing is to cover as much as possible from the total state space as long as the operational space contains no faulty area.

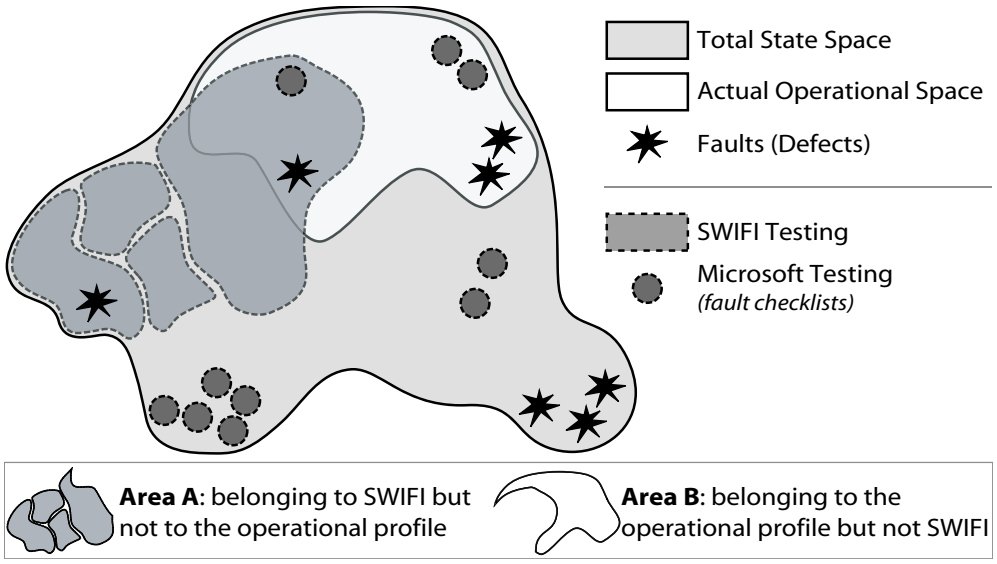

Figure 1.2: Total state space vs. operational space and coverage of various test paradigms

Various test tools cover the DD state space differently, the distinction being in the method used. While an in-depth discussion on existing test strategies is presented in Chapter 2, Figure 1.2 briefly illustrates the areas of the DD state space covered by such test tools.

For instance, *SWIFI* (SW Implemented Fault Injection – a popular paradigm used for testing robustness of SW components) injects faults into the program and then executes it using some workload in an effort to reach the injection points. Therefore, such a test strategy covers a part of the state space and this area is determined by the workload used. If this is divergent from the manner the DD executes in the field, then the adequacy of testing suffers. Such a situation is depicted in Figure 1.2, the area covered by *SWIFI* testing is not coincident with the operational space. Hence, while the *SWIFI* and the operational space subsets can intersect, two distinctions exist, namely (see lower part of Figure 1.2):

- **Area A: covered by SWIFI but not by the operational space.** Here the test resources were wasted, as the respective states are never reached in the operational mode;

- **Area B: covered by the operational space but not by the SWIFI.** The defects located in this area have high likelihood to be reached in the operational phases, though they are not covered by SWIFI.

In the light of this observation and under the assumption that relevant DD operational profiles can be captured, we conjecture that testing can be improved by primarily covering the operational space of the DD-under-test. Hence, the improvement to existing test paradigms is twofold. First, by minimizing the area A, the accuracy of the respective test method is enhanced and valuable test resources are saved. Second, by extending the test area into covering more of the area B, the adequacy of the test tool is enhanced as it addresses the subset of the test space actually reached in the operational phase by the DD-under-test. Subsequently, an ideal test tool would cover an area of the DD state space which exactly matches its operational space.

This observation is valid for other test paradigms beyond SWIFI. For instance, Figure 1.2 illustrates the test strategy employed by some DD test tools from Microsoft (used for acceptance test for Windows DDs) that exercise the state space. Knowing the usual mistakes made by DD developers, Microsoft builds and maintains pools of common fault patterns. Such pools are used to build fault checklists that the DD-under-test must pass before the respective DD can be delivered. Therefore, such tools test very small and precise subsets of the state space, creating situations when DDs containing non-standard defects (not present in the fault checklist) successfully pass the testing phase and are released for mass distribution. Microsoft tries to alleviate this effect by continuously updating its checklists and tools. Also, based on the failure reporting facility present in later Windows OSs, Microsoft collects and analyzes failures of many Windows computers worldwide, providing major DD development companies with feedback on how their DDs perform in the field. This allows for a timely distribution of patched or updated versions of the problematic DDs [Orgovan, 2008].

In order to translate the concept of *operational space* into an useful tool for improving existing test paradigms as described above, a procedural mechanism for constructing and distinguishing the operational space from the rest of the DD states is presented in this thesis.

Focusing on generating operational profiles to guide DD testing, our approach is based on monitoring the communication interfaces between the OS kernel and the DDs. At this interfaces, I/O traffic is captured and analyzed to build a state model of the DD. The state of a DD is represented by the set of DD functionalities observed to be in execution at specified instants. The transitions between states are triggered by incoming and outgoing I/O

requests (from a DD's perspective). The resulting behavioral model is used to discover execution hotspots in terms of frequently visited states and frequently taken DD code paths. Also, the introduction of operational mode quantifiers enables workload comparisons, by accurately measuring the similarity of the workload used in the lab against the actual DD workloads, as captured in the field.

Additionally, being non-intrusive and based on black-box principles, our framework is portable and easy to implement in DD profiling scenarios where the source code of OS kernel, workload applications or target DDs is not available. As an effort to validate the presented theoretical aspects, extensive empirical evaluations of actual Windows XP and Vista DDs are presented as case studies.

Overall, the work presented in this thesis focuses on providing a general operational state profiling framework for DDs. In addition, our work is an effort towards an improved DD test paradigm and not a test method *per se*. Developing a comprehensive stand-alone testing framework is not the intent of the current thesis though it is a subject of our ongoing research.

1.2 Thesis Targets and Contributions

1.2.1 Thesis Research Questions

The research questions driving the research presented in this thesis belong to two broad categories. The first one groups the conceptual definitions of the driver model and the model of the operational behavior of a driver, while the second one includes investigative questions regarding the experimental aspects.

Category I: Conceptual Questions

Research Question 1 (RQ1) *What are the events defining the operational state and progress?*

Chapter 3 sets up the model used to represent the operational activity of a DD. The model must observe the limitations of a black-box approach while not constraining its future enhancement with additional information obtained from source code insight. At the same time, the model must be general enough to be representative for as many DDs as possible, despite the inherent diverse nature of DDs.

Research Question 2 (RQ2): *Can the operational states be delimited from the rest of DD states?*

Chapter 4 raises and answers this research question, while discussing the various observed aspects, like the relation between the sizes of the operational state space and the total state space of a selected DD.

Research Question 3 (RQ3): *What are the aspects characterizing a visit of a state belonging to the operational state space?*
Chapter 5 discusses the state visit occurrence and duration aspects, as well as other facets of this problem from the testing perspective. The more often a operational state is visited, or the longer time is spent in the state, the more testing importance is associated with the respective operational state. High operational state occurrence indicates a possible execution hotspot, while a state where the DD spends longer time might indicate large amounts of code being executed and, therefore, a higher probability for a defect to hide in the respective operational state.

Research Question 4 (RQ4): *How different is the field execution of a DD from the test executions?*
The ability to accurately answer this question is a crucial condition for ensuring the adequacy of testing over the operational behavior of a DD. If the field execution differs considerably from the execution in the test lab, then the probability that still uncovered defects will surface and produce service failures is high, rendering useless all the efforts and resources spent for performing testing. Chapter 6 discusses these issues.

Research Question 5 (RQ5): *Can hotspots in the DD code be highlighted to help prioritize testing accordingly? What is the relation of the code-paths to the operational state model?*
Chapter 7 addresses these questions by introducing and then discussing the conceptual basis and the relevance for testing such *hotspots* (areas on the DD highly executed in the operational mode). If access to the source code is available, then the code-paths represent valuable information for other testing-related activities, like debugging for instance.

Category II: Experimental Questions

Research Question 6 (RQ6): *How can the DD activity be modeled in the absence of its source code? How can the operational mode of a DD be profiled and obtained experimentally?*
Further discussed in Chapter 4, these questions are answered by presenting the experimental methodology which was set up to monitor

and analyze the operational activity of DDs. The presented methods enable precise bordering of the operational states from the rest of the DD states.

Research Question 7 (RQ7): *How does one differentiate across the operational states of a DD?*
Chapter 5 introduces quantifiers for describing the operational mode of a DD. Based on the occurrence and temporal characteristics of each operational state, the quantifiers enable distinguishing among the states belonging to the operational mode of the DD.

Research Question 8 (RQ8): *How does one compare operational profiles of the same DD?*
Chapter 6 shows how the operational states are quantified and how the state quantifiers are used for differentiating among multiple operational profiles of the same DD. This activity is crucial for evaluating the degree of similarity between the test workload and the field workload in a quantifiable manner.

Research Question 9 (RQ9): *How can the code-path taken at runtime be inferred without access to DD's source code? How does one group similar DD code-paths into equivalence classes and which are the tradeoffs?*
Chapter 7 investigates these questions by monitoring and interpreting the information gathered on the functional interface of the DD, in addition to the I/O interface. Code paths are built and clustered together on relative similarity. Well established string similarity metrics are used to express the similarity between any two code paths and to define similarity classes.

1.2.2 Thesis Contributions

The research presented in this thesis makes several important contributions for the OS and testing research community. Listed below, the main contributions also enumerate the research questions they help answering.

Contribution 1 (C1) – Driver State Model: A DD model and an associated profiling technique via I/O traffic characterization inside the restricted-access OS kernel space are proposed. (RQ1, RQ2, RQ6)

Contribution 2 (C2) – Code Paths: A DD profiling technique based on the characterization of the OS–DD functional interface additionally to the I/O interface is presented and evaluated. This reveals the code

paths followed by the selected DD at runtime, helping the selection of test cases. (RQ5, RQ9)

Contribution 3 (C3) – Operational Profile Quantifiers: A set of occurrence- and duration-based quantifiers for accurate DD state profiling and workload characterization are introduced. They help distinguishing across the states of the operational profile, allowing for test prioritization. (RQ3, RQ7)

Contribution 4 (C4) – Execution Hotspots Highlighting I: An execution hotspot discovery method for black-box DD is presented. This is assisted by the ranked set of visited states and traversed transitions belonging to the operational state space of the respective DD. (RQ5, RQ7)

Contribution 5 (C5) – Execution Hotspots Highlighting II: An alternative expression of the execution hotspots is given in terms of the equivalence classes of the code-paths traversed in the operational phase. Such information helps debugging and also indicate locations where performance tuning is required (such hotspots are execution bottlenecks). (RQ5, RQ9)

Contribution 6 (C6) – Workload Comparison: A state-based methodology for accurate workload comparison is proposed. This enables the accurate quantification of testing adequacy, and also helps the choice of testing workloads. (RQ4, RQ8)

Contribution 7 (C7) – Extended Case Study: A large scale case study for Windows DDs was carried out, employing a large number of different DDs and workloads. The profiled DDs belong to several, current versions of Windows XP and Vista. (RQ6, RQ7, RQ8, RQ9)

Contribution 8 (C8) – Profiling Framework: A flexible and portable framework including tools for (a) monitoring the runtime activity, (b) constructing the operational profile of DDs and (c) call-path construction has been implemented and evaluated in our own and third-party experiments, validating its applicability and usefulness. (RQ6, RQ7, RQ8, RQ9)

1.2.3 Publications Resulting from the Thesis

The work reported in this thesis is supported by several international publications:

- **Constantin Sârbu**, Andréas Johansson, Neeraj Suri and Nachiappan Nagappan, *Profiling the Operational Behavior of OS Device Drivers*, submitted to the Empirical Software Engineering Journal, a Special Issue for ISSRE 2008 best four papers (in review), 2009

- **Constantin Sârbu**, Nachiappan Nagappan and Neeraj Suri, *On Equivalence Partitioning of Code Paths inside OS Kernel Components*, in Proceedings of the First International Workshop on Software Technologies for Future Dependable Distributed Systems (STFSSD), Tokyo, 2009 (to appear)

- **Constantin Sârbu**, Andréas Johansson, Neeraj Suri and Nachiappan Nagappan, *Profiling the Operational Behavior of OS Device Drivers*, in Proceedings of 19th International Symposium on Software Reliability Engineering (ISSRE), Seattle / Redmond, pp. 127 – 136, 2008

- **Constantin Sârbu** and Neeraj Suri, *On Building (and Sojourning) the State-space of Windows Device Drivers*, State-space Exploration for Automated Testing Workshop (SSEAT), Seattle, 2008

- **Constantin Sârbu**, Andréas Johansson and Neeraj Suri, *Execution Path Profiling for OS Device Drivers: Viability and Methodology*, in Proceedings of the 5rd International Service Availability Symposium (ISAS), Tokyo, in Springer Verlag's LNCS 5017, pp. 90 – 109, 2008

- **Constantin Sârbu**, Andréas Johansson, Falk Fraikin and Neeraj Suri, *Improving Robustness Testing of COTS OS Extensions*, in Proceedings of the 3rd International Service Availability Symposium (ISAS), Helsinki, in Springer Verlag's LNCS 4328, pp. 120 – 139, 2006

1.3 Thesis Structure

The structure of the following chapters follows the structure of the research questions described earlier:

Chapter 1 presents the background of the problems driving this research, introduces the research problems and the contributions of this thesis.

Chapter 2 introduces the terminology used throughout the thesis and surveys the state of the art and practice in the field of validation and verification of OS and DDs.

Chapter 3 presents and discusses the system model and the OS–DD communication models used throughout this thesis. The model used for describing the runtime activity of a DD is also introduced in this chapter.

Chapter 4 defines and presents various aspects related to the model of the operational space and its characteristics.

Chapter 5 defines the quantifiers used to describe the operational modes of a DD.

Chapter 6 shows the impact operational mode quantifiers introduced in Chapter 5 for enabling cross-comparisons among operational profiles.

Chapter 7 presents and investigates a methodology that highlights the paths followed through the DD code, when only the OS–DD interfaces are monitored (without source-code level access).

Chapter 8 finally concludes the thesis, re-evaluating the value of the conceptual and experimental contributions. A discussion on the applicability of the thesis results to different fields of DD testing is provided, alongside with an outline of the future research directions opened by the novel approach presented by this thesis.

Chapter 2

State of the Art and Practice

Why is defining the operational space a critical prerequisite for software verification and what are the current uses of the operational profiles?

Generally, *fault-tolerance* is the ability of a system to provide its specified services even in the presence of failures of one or several of its components. A component is considered fault-tolerant if it can cope with internal and external failures (that is, failures of its internal parts and failures of the neighboring components). Fault-tolerance is not a binary concept. In the presence of failures, some systems are able to provide their services at a degraded level, a situation called "*graceful degradation*". Fault-tolerance is a property particularly sought-after in high-availability, life- or mission-critical systems but as OSs have become ubiquitous, their tolerance to faults (or defects / bugs) is an increasingly desired attribute.

As an important basis for the context of the research presented in this thesis, this chapter starts by discussing the concept of fault-tolerant computing and related aspects. Onwards, different validation and verification paradigms and tools used in the design and engineering of fault-tolerant systems (and OS components) are surveyed. Brief discussions on test space aspects accompany each of the presented techniques. The chapter concludes with a discussion on several paradigms for operational mode profiling, alongside with their most prominent applications, both in industrial and academic environments.

2.1 OSs' Faults, Errors and Failures

The terms *faults*, *errors* and *failures* are key concepts in the design of fault-tolerant systems. Being intensively used in different fields of research, these concepts bear different names. For instance, in SW testing they are collectively called "*bugs*". The slight differences and the generally-accepted translations in five different languages are tackled in the book "*Dependability: Basic Concepts and Terminology*" [Laprie, 1992]. The article on "*Basic Concepts and Taxonomy of Dependable and Secure Computing*" defines and explains the nuances among the three concepts [Avižienis et al., 2004]. As the work presented in this thesis adheres to the definitions given by Avižienis et al., we briefly introduce *faults*, *errors* and *failures* as presented in the mentioned article.

Faults are anomalies resulting in service deviations (errors). By exercising the system, a fault can be activated, thus becoming an *error*. An error propagates to the outputs of the system and causes a *failure* (service disruption). As usually the user of the system primarily observes its outputs, it is generally considered that a failure is the manifestation of an activated fault.

In component-based systems (often the situation for modern computing systems), failures spread from one component to another, leading to long propagation chains. This effect considerably hampers the locating and subsequent fixing of the original fault (this process is commonly called "*debugging*").

The error propagation in static SW components was formalized in the EPIC framework [Hiller et al., 2004], using the PROPANE tool as a study environment [Hiller et al., 2002]. Johansson et al. redefined the EPIC measures for OSs and used them to highlight and characterize the propagation paths allowing faults in DDs to reflect at the user interface in Windows CE [Johansson et al., 2004]. Subsequently, the insight gained into the propagation paths was used to improve the robustness testing of Windows CE DDs, by addressing key aspects related to the effects of error *location* [Johansson and Suri, 2005], *type* [Johansson et al., 2007b] and *activation instant* [Johansson et al., 2007a].

It is important to note that often the fault-tolerance of a system is defined in terms of the nature of the faults that the respective system can cope with. Fault-tolerance is not generic, a system is only tolerant to the faults that it was designed to cope with (under the assumption that its fault-tolerance mechanisms perform in a correct fashion). Choosing an example from the field of interest of this thesis, a DD which is tolerant to faults of the managed hardware device is not necessarily tolerant to faults manifesting themselves in the DD's communication interface with the OS kernel. Hence, when dis-

cussing the fault-tolerance of a system, the fault types which the respective system is tolerating must be properly specified.

A large body of research on design of fault-tolerant systems is dedicated to precisely defining faults as a precondition for enforcing tolerance to the respective faults, called *fault modeling*. While initially the main focus was on hardware faults, the advent of SW increased the relevance of SW faults as source of failures as high as 60% of the faults being SW faults in early 1990's Tandem systems [Gray, 1990]. Tandem systems used a number of redundant processors and storage devices to provide high-speed fail-over in case of a hardware failure.

As fault-tolerance mechanisms able to cope with all possible faults are highly desirable but unfortunately unattainable, early identification of the perturbations that the system will be subject to in the field is crucial. In an effort towards identifying system failures, large companies collect usage data about their products. Unfortunately, this activity is subject to extensive discussions considering the legal and ethical consumer privacy concerns. Microsoft introduced the "Customer Experience Improvement Program" [Murphy, 2004; Orgovan, 2008] collecting failure reports from computers running Windows OSs worldwide. The collected information is based on a customer participation choice and analyzed to better understand the field failures. The results are subsequently used to improve the fault-tolerance of existing and future versions of the OS components and applications responsible for the reported failure.

As the OS acts as a mediator between user SW and hardware devices, it has to cope with errors coming from both neighboring levels and also from the various OS-internal components, such as DDs. Following an 1985 study on Tandem outages, Gray reported four main sources for the observed failures: hardware, software, administration and environment [Gray, 1985]. In Microsoft's Windows NT, Xu et al. found in 1999 that the main reboot causes were maintenance activity (31%), SW (22%) and hardware failures (10%) [Xu et al., 1999]. In the following, we present each of the main classes of faults affecting the stability of current OSs.

Hardware-related Faults

Hardware-related failures of an OS originate in the physical faults contained in the hardware of the computing system of which the respective OS is part of. The hardware failures have *internal* causes (i.e., production defects, power transients, aging of the electronic parts, etc.), are the effects of *external* interference (i.e., radiation, electro-magnetic fields, etc.) or penetrate the system via its *interfaces* (i.e., noisy input lines) [Avižienis et al., 2004].

To cope with hardware failures, different approaches were taken, at different levels of the computing system. To contain (as much as possible) the errors generated by malfunctioning hardware to propagate to other hardware or SW parts of the system (for instance to the OS or user applications), the hardware itself was enhanced with error detection and correction mechanisms. Redundancy was intensively used for this purpose in different forms: as *data redundancy* (e.g., cyclic redundancy check (CRC) codes), as *spatial redundancy* (e.g., redundant communication lines, duplex systems and TMRs) and as *temporal redundancy* (e.g., data re-transmissions).

Kao et al. identified several classes of hardware faults and characterized them from the OS perspective into four classes [Kao et al., 1993]:

1. *Memory faults* – corruptions of memory cells, affecting both the code and data regions of the programs loaded in memory;
2. *CPU-related faults* – corruptions of CPU registers, also the control flow of running programs was affected, due to corruptions in the PC register;
3. *Bus faults* – corruptions of the various bus lines and the communication using the affected lines;
4. *I/O faults* – corruptions of the hardware peripheral devices.

Another key characteristic of the hardware faults is their persistence. Some are *permanent* that is they remain in the system as long as the hardware part replaced, but some are *transient*[1] their effects completely disappear after some time.

When the fault-containment mechanisms of the hardware parts of the computing system fail to restrict the failure propagation to the hardware boundaries, hardware faults sometimes propagate to SW (and OS components). When a hardware error propagates all the way up to the user applications, the source of the failure is usually difficult to infer due to the long propagation path, involving both hardware and SW components.

Software-related Faults

Different in nature from the hardware faults, the SW-related faults increasingly attracted the attention of the fault-tolerance community as the role of SW in computing activities started to grow. While initially the value of SW

[1]Permanent faults are often called "*Bohrbugs*" – bugs manifesting themselves consistently, transient faults are called "*Heisenbugs*" – their manifestation is erratic. Both Bohrbugs and Heisenbugs can also manifest in SW [Gray, 1985].

programs was considered insignificant in computers when compared to the importance of hardware, the balance changed in favor of SW as soon as the first OSs appeared. Due to much lower production costs, many hardware mechanisms moved to SW thus also enabling a finer degree of control of the computer to its users. While the production of hardware parts reached a certain maturity and thus stability, the SW counterpart witnessed a widespread development. Soon, the SW defects were reported to cause more system failures than the hardware (60% of total failures were caused by SW in the beginning of the 1990's [Gray, 1990]).

In an effort to characterize SW faults under the IBM's MVS OS "*in order to gain the insight needed to provide a clear strategy for avoiding or tolerating them*", Sullivan and Chillarege classified the sources of SW failures, distributing them into two large groups [Sullivan and Chillarege, 1991][2]:

1. *Regular* – are the "typical software errors encountered in the field" as type mismatches, uninitialized pointers, synchronization errors, statement logic etc.;

2. *Overlay* – are "defects that corrupt a program's memory" spanning boundary conditions, memory allocation management errors, invalid pointers, timing errors.

Overlay errors were found to have much larger impact than the regular defects as they are, by nature, harder to find and fix. Sullivan and Chillarege also observed that (a) boundary conditions and allocation errors are the most common defects and (b) most overlays manifest themselves as corrupted data in small memory locations close to the data intended by the memory update.

In a later study that considered two large database systems from IBM (additionally to the MVS OS), Sullivan and Chillarege compared defect and error type, and distributions of error activations under the three mentioned SW products [Sullivan and Chillarege, 1992].

Further summarizing the observed characteristics of the SW defects, Chillarege et al. developed a set of seven defect types as the seminal *Orthogonal Defect Classification* (ODC) process [Chillarege et al., 1992]. According to ODC, the SW defect types are: function, assignment, interface, checking, timing/serialization, build/package/merge and documentation. Used as a common reference platform in the SW testing community, many subsequent test approaches used the ODC classification to formally specify the targeted defects (for instance, Durães and Madeira [2006]; Johansson et al. [2007a,b]).

[2]In their work, Sullivan and Chillarege refer the SW defects as "errors".

User-related Faults

Xu et al. found that the user-related faults represented 31% of total outages of a network of servers equipped with Windows NT [Xu et al., 1999]. These faults were related to improper system configuration and planned maintenance. System configuration faults are defects generated by inexperienced system administrators, while planned maintenance defects were added to the system by applying SW updates or by adding patches to existing SW or OS components. Subsequently, Murphy and Levidow reported similar results [Murphy and Levidow, 2000], additionally indicating DDs as a significant source of outages.

The Customer Experience Improvement Program (CEIP) was introduced by Microsoft to collect rich reliability data from computers running Windows [Orgovan, 2008]. CEIP allows Microsoft to obtain failure rates and failure prevalence from hundreds of thousands Windows Vista computers. According to the CEIP published results, application and DD install procedures are in the top nine sources of Windows Vista failures.

Driver-related Faults

As DDs mediate the communication between user SW applications and the hardware devices, they have to cope with failures coming from both directions. For instance, a faulty user application might call in a DD in an invalid fashion or send it invalid parameters. A faulty hardware device might send its associated DD malformed or incorrect data. In addition to being tolerant to faults coming from external sources, DDs also need to handle their internal operational defects. Coping with faults from multiple sources and belonging to multiple types requires DDs to contain complex fault-tolerant mechanisms.

Unfortunately, as DDs perform their activity as part of the I/O chain (notoriously slow) and inside the OS kernel (where performance is vital for providing responsiveness), adding mechanisms to detect and handle errors at runtime is problematic. Johansson et al. studied the error propagation mechanics for Windows CE .Net DDs [Johansson and Suri, 2005], observing that DDs are susceptible to propagating errors leading to severe OS failures.

As DDs are currently add-on components of the OS kernels, they represent large parts thereof (about to 70% of Linux code [Swift et al., 2005]). Chou et al. observed that current DDs have error rates three to seven times higher than other OS components [Chou et al., 2001]. Interestingly, the same source revealed that some DD-related functions contain more defects than others, and that newer files have more defects than older ones. This observation

confirms the research from the area of general SW testing that found that defects tend to cluster [Möller and Paulish, 1993].

Other authors like Ganapathi and Patterson; Ganapathi et al.; Murphy and Levidow also found DDs to be the main source of OS failures [Ganapathi and Patterson, 2005; Ganapathi et al., 2006; Murphy and Levidow, 2000].

The Customer Experience Improvement Program (CEIP) developed at Microsoft revealed that different types of DDs account for most of the top 500 Vista SP1 OS crashes. Table 2.1 lists their results, as of September 2008.

Table 2.1: Top 500 Vista SP1 OS crashes for September 2008 (adapted from Vincent Orgovan's keynote talk at ISSRE 2008, see Orgovan [2008])

Category	**% Crashes**	**Notes**
Networking	12.5	Mostly power management
Display	9.4	Mostly video cards
OS Core	8.6	Kernel 3.3%; USB 3.3%
Application drivers	6.5	Antivirus 3.6%; Malware 1.1%; Firewall 0.5%
Hardware	6.3	General 2.7%; Memory 2.2%; Disk 1.4%
Triage	5.7	Not well classified as of November 2008
Corruption	5.1	Cannot be classified
Storage	5.0	Mostly RAID controllers, some IDE/API
Peripherals	2.6	Mostly personal media players
Imaging	1.7	Camera drivers, USB video
Streaming Media	0.8	Third-party cameras and TV tuners
Audio	0.6	Audio cards and HD drivers
Input	0.5	Third-party mice
Issues Beyond Top 500	34.6	Haven't looked at many of these

Without being too precise[3], Table 2.1 indicates that most of the Vista SP1 crashes are explicitly related to DDs, some are due to DDs not being able to cope with failures of the underlying hardware devices, and some are generated by hardware defects.

2.2 Verification and Validation Techniques

In the process of developing fault-tolerant systems an important aspect is to determine if if the correct system is designed (*validation*), and if the system is correctly designed (*verification*).

This section briefly presents the main current verification and validation techniques for SW components. It introduces the background and terminology as a basis for the more detailed discussion on the state of the art and

[3]Microsoft cannot precisely pinpoint the faulty SW and hardware components as they usually belong to third-party producers.

state of the practice in validation and verification of DDs which is developed in Section 2.3.

2.2.1 Verification and Validation of Fault-tolerant SW Systems

To determine if a SW product meets its requirements (validation) and if the outputs of the design phases are correct (verification) numerous paradigms and tools have been developed. Such approaches are used in the development phases of many SW products, but they are usually applied for fault-tolerant systems. Without attempting an exhaustive discourse on the base topic of verification and validation (V&V), we briefly present commonly used techniques in academia and industry.

The verification process encompasses both static and dynamic steps. That is, often the source code is first inspected (static) and then it is executed against specific test cases (dynamic). In contrast, validation is done dynamically. For instance, the SW product is (symbolically) executed in typical and atypical use scenarios.

Some V&V techniques cross the theoretical boundaries between validation and verification in an attempt to constitute complete solutions for ensuring the product's fault-tolerance. Hence, we list below the most common validation and verification approaches, without attempting to strictly classify them either as validation or verification methods:

- *Monitoring and measurements* – the behavior of the system is monitored in the operational phase and specific measurements are carried out to quantify various fault-tolerant attributes of the system;

- *Simulation* – the fault-tolerant attributes of the SW system (or protocol) are validated using an artificial environment that emulates field installations;

- *Formal verification* – includes theorem proving, model checking and static code analysis; a model of the system (or protocol) is first abstracted and then used to verify its properties, further discussed in Section 2.2.2;

- *Testing* – is performed at different levels (functional, structural, etc.) and, according to Myers consists in "*executing a program with the intent of finding errors*" [Myers, 2004]; testing is further discussed in Section 2.2.3;

- *Fault injection (for hardware or SW)* – faults are artificially inserted into the system with the purpose to validate the correct functioning of the fault-tolerance mechanisms, is further presented in Section 2.2.4;

The approach presented in this thesis primarily falls into the "*Monitoring and measurements*" category, and it is intended as a supporting tool for existing testing and SW fault injection paradigms. It uses a model of the DD and develops a state diagram of the operational phase, thus being similar in some respect to the abstracted system model used by formal verification methods. To highlight the relation to the mentioned V&V techniques, we briefly discuss them in the following sections.

2.2.2 Formal Verification

In formal verification a model of the system alongside with powerful mathematical techniques are used to express the system's properties, which are then verified against its formally expressed specifications.

Formal verification uses multiple techniques to analyze the behavior of the modeled system. For instance, *theorem proving* is concerned with proving that the implementation follows its specification. In theorem proving the model of the system is expressed as axioms using some mathematical logic (for instance, λ–*calculus*) and theorems are inferred using a set of "deduction rules". Theorem proving is a powerful technique, but it is difficult to use because of the task of manually specifying the deduction rules, thus relying on the expertise of the person using it.

Another widely spread formal verification technique is *model checking* [Clarke et al., 2001], which improves on theorem proving by enabling a higher degree of automation. In model checking the system is modeled using a state transition diagram (*Kripke structure*) which is then automatically and exhaustively explored by the model checker. Unfortunately, model checking often suffers from state space explosion[4] problems, thus their value being currently limited to validation of small size systems and protocols. While attempts to reduce the state space of larger and complex systems and protocols such that they can be handled by model checkers have been done, they often cannot claim checking completeness against the actual system as only a subset of the possible states are covered.

Static code analysis is a method performed without the actual execution of the program with the purpose to identify violations (or inconsistencies) of

[4]State space explosion manifests itself as a too large state space (all states plus the transitions among them) to be exhaustively covered in useful time, when the model has a high detail level.

the specifications [Nielson et al., 2005]. The analysis is automatically performed by a tool which takes as input either the program's source code or object code. The sophistication of the performed analysis ranges from individual program statements to complete programs containing multiple source code files. Static code analysis usually highlights coding errors (e.g., the *Lint* tool – Johnson and Johnson [1978]) or mathematically prove program properties (the *SDV/SLAM* tool – Ball et al. [2004]). Hayes and Offutt have used static analysis to infer test cases for helping subsequent testing of an large-scale industrial SW (Tomahawk Cruise Missile) [Hayes and Offutt, 2006].

Formal methods represent a promising approach for systematic and automatic verification in the context of the current complexity of SW programs. Unfortunately, their usage is limited to verification of mission- and safety-critical systems, as their usage usually requires expertise (theorem proving) or suffer from state space explosion (model checking). More applicable, the static code analysis used by the *SDV/SLAM* for the formal verification of Windows DD is discussed in detail in Section 2.3.

2.2.3 Software Testing

In SW testing, *test cases* are executed against the program-under-test with the intent of revealing faults present in the code [Kaner et al., 1999; Myers, 2004]. Test cases define test context (i.e., which part of the program is targeted, the actions that need to be performed to reach the desired part, etc.), test inputs and the expected outcome (as defined in program specifications). After the test is executed, the actual outcome is compared with the expected one. In the case of identity, the respective test case is said to be *unsuccessful* (the testing did not manage to reveal a defect).

Many approaches for testing exist, and various test paradigms were developed for being used in various SW development phases, for testing different attributes of the software (i.e., usability, security, performance, reliability, documentation, acceptance). Depending on the choice if the internal structure (i.e., source code) of the SW is used or not in the testing process, SW testing is considered to belong to two large classes, namely *black-box* and *white-box* testing.

In *black-box* testing (alternatively called "data-driven" or "input/output-driven" testing) the program is viewed as a black box, i.e., the internal structure is not utilized. Instead, the test process focuses on checking if the outputs follow specifications for each possible combination of inputs. To ensure test completeness, the full parameter range of each input line has to be tested, and also all possible combinations of inputs, too. If one tests

for robustness, values lying outside the specified range of inputs have to be tested as well.

Unfortunately, exhaustive input testing is hard to attain, as the duration of the test procedure might be too long to be useful, even when a highly automated tool is used. Moreover, some programs use internal variables to decide the control flow (i.e., *if* branches on internal counter variables). In such cases black-box testing, being unaware of the internal program structure, might report erratic outcomes when the same inputs were repeatedly fed to the system.

In contrast, *white-box* testing (also termed "logic-driven" testing) considers the program's source code as an important source of information about the structure of the program. In principle, the access to the program's internal structure might suggest that complete coverage is achievable, as constructing relevant test cases is easier. While reaching 100% code coverage seems hard if not impossible [Kaner et al., 1999], several levels of code coverage were discussed in the testing community in an attempt to establish when is the right time to stop testing [Dalal and McIntosh, 1994; Huang and Lyu, 2005; Huang and Boehm, 2006; Musa and Ackerman, 1989]. Such examples include *statement coverage* (all statements of the code must be tested), *branch coverage* (all decision branches have to be taken) and *path coverage* (all paths in the program must be followed).

In his book on testing, Myers showed that the path coverage (the strongest coverage strategy out of the previously mentioned ones) is infeasible even if the program contains only finite loops. He used as an example a "*10-20 statement program*" having few loops and branches that created approximatively 10^{14} possible paths. If one needs one test case to cover each program path, and each test case takes one second to be executed, then the complete path testing of the respective program would last about 3.2 million years, much too long to be useful.

Despite the mentioned test space coverage issues, testing is currently intensively used in both industry and academic environments as the main V&V strategy for both hardware and SW. Using it, a certain level of assurance for a desired operational requirement can be built. Unfortunately, the success of testing (that is, showing that the program contains defects) is a two-edged sword: on one hand, it indicates where bugs are thus helping their removal, and on the other hand it proves that more bugs exist in the code. As full coverage is usually not achievable, claiming that all the bugs were found is not founded. Similarly, not finding bugs does not prove their absence, but merely shows the used test method is not appropriate. Edsger Dijkstra considered that testing "*is a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence*".

Testing is a powerful V&V technique which suffers from two key problems: coverage issues and construction of suitable test cases. Both determine the effectiveness of testing in an decisive manner. The approach presented in this thesis helps testing from different perspectives. It accurately *defines* the operational test space which needs to be covered, *guides* test case choices and *prioritizes* testing onto the code areas mostly activated in the field.

2.2.4 Fault Injection

The main purpose of *fault injection* is to identify the robustness weaknesses of the targeted system. For this purpose, artificial faults are injected to activate conditions that trigger the fault-tolerance mechanisms present in the system (error detectors and error correctors).

Initially, fault injection started as a technique for evaluating the dependability of hardware components [Arlat et al., 1993]. Physical, hardware probes were inserted into the hardware components to inject faults and to read the outcomes. As more access to the hardware became available via the SW part of the system at cheaper costs than the physical implementation, a new breed of fault injection techniques appeared, termed *SWIFI* (Software implemented Fault Injection) [Hsueh et al., 1997].

Due to its flexibility, in addition to testing robustness of SW systems, *SWIFI* was also used to verify security mechanisms [Chen et al., 2002; Neves et al., 2006]. *SWIFI* was also used for OS robustness evaluation. The *Ballista* project developed a robustness benchmark for the POSIX interface of the UNIX/Linux OSs to user applications [Koopman and DeVale, 1999, 2000]. *SWIFI* was also used for evaluations of the performance of OS I/O libraries [DeVale and Koopman, 2001], CORBA services [Pan et al., 2001] or Win32 libraries of Windows OSs [Shelton et al., 2000].

Fault injection was used also as a prerequisite for the automatic construction of robustness wrappers. For instance, *HEALERS* [Fetzer and Xiao, 2002a,b] uses an adaptive fault injection mechanism that analyzes the parameters of C library functions using the information provided in the header files and manual pages. Based on this, fault injectors are constructed to obtain the *robust argument type* (the set of parameter values for which the tested function performs correctly). The type domains obtained are utilized to generate robustness wrappers for the respective functions. The *Autocannon* tool [Süßkraut and Fetzer, 2007] developed later *HEALERS* using a more comprehensive type system adapted from the *Ballista* project. Also the *AutoPatch* project [Süßkraut and Fetzer, 2006] is based on *HEALERS*, and focuses on how applications handle the error codes provided by library functions. Using fault injection to return error codes from the targeted functions, the unsafe

functions are highlighted.

Fault injection and *SWIFI* are related to testing as their primary goal is to find robustness weaknesses as opposed to finding defects (which is the goal of testing). Therefore, both face the same problems of coverage and test case selection as testing does. Additionally, fault injection (and *SWIFI*) has to determine which faults to inject, a difficult problem in itself.

2.3 Current Verification of OS and Device Drivers

The main focus of this thesis is on providing guidance to existing V&V techniques for OS components and, in particular, for DDs. This area of research is receiving an increasing amount of attention in the testing community, attempting to parallel the widespread proliferation of COTS OS outages. The used paradigms for testing OS components employ the V&V techniques previously surveyed and also hybrid ones, in order to cope with the characteristics of the OS components.

An important step in producing SW programs is the compilation stage. According to Aho et al. "*a compiler is a program that reads a program written in one language – the* source *code – and translates it into an equivalent program in another language – the* target *language*" [Aho et al., 1986]. Usually, the target language is the machine language understood by the hardware platform for which the source program is compiled for. As an important part of this translation process, the compiler reports the presence of errors in the source program. In this respect, a compiler can be considered a basic tool for finding lexical, syntactic or typos for the language of the source program. We consider that the discussed OS components are fully "compilable", that is, they contain no errors that a compiler can find.

From the viewpoint of the discussion presented in this section, the compilation stage only helps differentiating among the various existing V&V techniques for OS components, and is not necessarily considered a verification method *per se*. In this respect, we distinguish two main directions in V&V for OS components. The first one exclusively uses the *source* code of the respective OS component, while the second one uses the *target* code of the component for the same purpose.

Thorough testing throughout all development phases is key to creating robust, fast and efficient OS components. Finding and fixing as many defects, as early as possible in the development life-cycle is critical to reduce the likelihood that failures of the component will cause system outage in the

field, after releasing the respective OS component to customers.

Therefore, the rest of this section discusses the state of the art and practice in the area of OS component V&V, useful *at compile time* (described in Section 2.3.1) and *at runtime* (described in Section 2.3.2). Some approaches that isolate the components found to be faulty from the rest of the OS are also discussed in this section.

2.3.1 OS Component Verification At Compile Time

As the DDs for the latest additions to the Windows OS family (XP and Vista) are considered for a case study in this thesis, we focus here on the testing tools provided by Microsoft. To help the development of DDs for its OSs, Microsoft specifies the architecture and the interface that a DD must follow to function properly in Windows OSs.

Prior to Server 2003 and Vista, the DD specifications were termed as *Windows Driver Model* (WDM - Dekker and Newcomer [1999]; Oney [2003]). To support WDM, Microsoft provided alongside with it a set of documentation, development and testing tools – the *Driver Developer Kit* (DDK). As the model used for drivers was slightly updated in Vista (user-mode drivers were added), WDM was renamed to *Windows Driver Framework* (WDF), and DDK to *Windows Driver Kit* (WDK) [Orwick and Smith, 2007].

WDK contains tools testing tools for DDs falling into two categories: (a) *static code analysis tools* which are described in this section and (b) *dynamic verification tools*, further presented in Section 2.3.2.

PRE*f*ast is a source code analysis tool that main goal is to find coding errors that cannot be found by a compiler [Orwick and Smith, 2007, Chap. 23]. Using the DDs source code, PRE*f*ast builds the set of possible code paths and then simulates their execution. The paths are checked against a (manually specified) set of rules designed to highlight errors and bad coding practices which are then logged as warnings. The classes of errors PRE*f*ast can detect are listed in Table 2.2.

While being a powerful tool, PRE*f*ast requires proper expertise to be efficiently used. The primary problem is handling the large amount of noise (false-positives) present in PRE*f*ast's log files. This is handled by using predefined filters or by inserting annotations into the source code which is fed to PRE*f*ast. For large and complex DDs this activity becomes a considerable overhead for developers. Moreover, mistakes can occur in the process of adding annotations or writing log filters, thus leading to false-negatives (actual errors which are not reported as such).

Secondly, executing *all* code paths is often infeasible, due to their large number originating in the code complexity. Hence, to keep PRE*f*ast's execu-

Table 2.2: Classes of errors detected by PRE*f*ast (adapted from [Orwick and Smith, 2007, Chap. 23, pp. 733])

Category	**Errors tested for**
`Memory`	Memory leaks, uninitialized memory, de-referenced NULL pointers, etc.
`Resources`	Failures to release used resources such as locks
`Function Usage`	Function argument type mismatches, use of obsolete functions, etc.
`Floating-point`	As the support for floating-point operation is limited in some hardware devices, the floating point state has to be correctly handled in the DD
`Precedence Rules`	Situations when DDs do not function correctly due to C language precedence rules
`Kernel-mode Coding`	Errors occurring when the kernel-mode DD does not follow the coding practices recommended in the WDF, such as using a user-mode string library instead of its kernel counterpart
`Driver-specific Coding Practices`	Specific operations which are often source of errors in DDs, such as wrong initializations of DD-specific routines

tion time in manageable limits, the upper bound for the number of exercised paths must be specified when PRE*f*ast is started (via the `/maxpaths` command line option). This reduces the effectiveness of PRE*f*ast in covering relevant parts of the DD's state space.

Static Driver Verifier (termed *SDV* henceforth) is another verification tool for DDs which is designed to be run towards the end of the development cycle, after the defects found by PRE*f*ast were fixed. *SDV* started as a research program at Microsoft Research (MSR), intending to create an automatic engine to verify if a supplied C program correctly uses the interfaces to its external libraries [Ball and Rajamani, 2002; Ball et al., 2004, 2006]. The project – called *SLAM* at MSR – was successful in lab scenarios and made the leap to the DD's verification practice: its engine was incorporated in *SDV* [Knies, 2005] and made available to DD developers as a part of DDK and later, WDK [Oney, 2003] [Orwick and Smith, 2007, Chap. 24].

SDV/SLAM used *Bebop* [Ball and Rajamani, 2000], a symbolic model checker that automatically explores the state space of the targeted DD. To specify the DD's interfaces with the OS kernel's API libraries, Ball and Rajamani developed and used the *SLIC* formal language [Ball and Rajamani, 2001]. The main categories of *SLIC* rules are listed in Table 2.3.

SDV symbolically executes the source code of the DD against an own, simplified model of the OS. The OS model contains several worst-case scenarios, such as continually failing system calls. In this "hostile" environment,

Table 2.3: Rules tested for in *SDV* (adapted from Microsoft Corp. [2009d])

Category	Rules tested for
`IRP`	Functions that use I/O request packets
`IRQL`	Functions that use interrupt request levels
`PnP`	Plug and Play functions
`PM`	Power management
`WMI`	Functions that use Windows Management Instrumentation
`Sync`	Synchronization related to spin locks, semaphores, timers, mutexes, and other methods of access control
`Other`	Functions that are not fully described by any of the other categories

SDV tries to prove using *Bebop* that the DD violates a set of interfacing rules to the OS (defined using *SLIC*). If no rule violation is found, the DD passes the *SDV* verification.

An abstract model of the DD is used instead of the real DD in order to keep the number of DD states within manageable limits for the model checker, inducing a limitation of the coverage for the real DD. As a realistic limitation *SDV* is used with DDs having less than fifty thousands lines of code, and it is currently limited to WDM and kernel-mode WDF DDs [Orwick and Smith, 2007, page 824]. Recently, the *SLAM* engine was replaced with *SLAM v2.0*, in which the *Z3* theorem prover is used instead of *Bebop* [SLAM2.0, 2009]. The future will show the effectiveness of the new *SDV* engine over the older one.

2.3.2 OS Component Verification At Runtime

Binary Instrumentation

While the proportion of lines of kernel code corresponding to DDs is on the rise mostly due to higher OS support for peripherals [Swift et al., 2005], the need for novel DD-specific testing approaches and tools is also increasing. As a consequence, various paradigms were successfully applied to testing OS code and in particular to DDs. Notable runtime testing tools include *PURIFY* [Rational Inc., 2009] and Microsoft's *Driver Verifier* [Microsoft Corp., 2009a]. Such tools instrument a binary image with a set of checks, thus enabling the examination of the respective binary's runtime behavior. The checks are designed to detect illegal activity of the instrumented binary, avoiding potential system corruption.

Specifically designed for Windows DDs, *Driver Verifier* is included in all releases of Windows 2000, XP, Server 2003 and Vista and continually enhanced with each new version of the Windows OS [Orwick and Smith,

2007, Chap. 21]. *Driver Verifier* traps at runtime many error conditions that might go unnoticed (being masked by the respective DD or by the OS), and signals the error condition by immediately stopping the system and supplying the error details in the form of a "blue screen". To ease the finding of the defect that caused the error condition, a memory dump is saved for the later use with a kernel debugger tool.

Driver Verifier performs on DDs which are installed and running in the system. It intercepts the system calls made by the DD, injects them with faults and then waits until an error occurs. Also, *Driver Verifier* can simulate extreme conditions for the DD's environment, by limiting the available resources (memory, locks, timers, etc.). *Driver Verifier* can be configured to monitor the activity of multiple DDs at a time. Also the conditions that are checked can be configured, by selecting them from a list. Table 2.4 briefly lists the conditions checked by *Driver Verifier*, as its latest version released with Windows Vista.

Table 2.4: *Driver Verifier* options (adapted from [Orwick and Smith, 2007, pp. 678])

Category	**Rules tested for**
`Deadlock Detection`	Use of spin locks, mutexes, etc.
`Disk Integrity`	Monitors HDD activity and data preserving
`Direct Memory Access`	Use of DMA routines, buffers and adapters
`Driver Hang`	Times DD routines to report passing given deadlines
`Enhanced I/O Verification`	Performs stress testing of DD's I/O routines
`Force IRQL Checking`	Detects access to paged memory at wrong interrupt levels
`Force Pending I/O requests`	Tests response to STATUS_PENDING return values
`I/O Verification`	Illegal or inconsistent use of I/O routines
`IRP Logging`	Logs the use of I/O request packets (IRP)
`Low Resources Simulation`	Randomly fails memory allocation requests
`Miscellaneous Checks`	Checks for common causes of DD crashes
`Pool Tracking`	Checks for memory leaks
`Special Pool`	Monitors memory requests for over- and under-runs

The major weakness of the runtime instrumentation tools resides in the dependency on the quality of the checks and on the activated parts of the tested DD. That is, if a condition goes unchecked, no corruption is signaled. To alleviate this issue, Microsoft encourages the use of *Driver Verifier* at all development stages, especially in the testing phase. To be effective in finding defects in DDs, *Driver Verifier* should be active when other DD test tools included in WDK are used.

SWIFI for OS Components

Software-implemented Fault Injection (*SWIFI* – see Section 2.2.4) was often used to test the robustness of various OS components [Koopman and DeVale, 2000; Pan et al., 2001; Shelton et al., 2000]. Though, only a few research efforts have addressed the DDs as targets for robustness assessments.

Durães and Madeira mutated the binary images of Windows 2000, NT4 and XP floppy disk and CDROM DDs by injecting single faults and studied the outcomes in terms of OS availability, stability and user feedback [Durães and Madeira, 2002a]. The executable is scanned for certain machine-code patterns, and mutations are performed using the patterns in order to emulate higher-level faults. While the approach is useful as a case study for the injection mechanism (further presented in Durães and Madeira [2002b]), the presented results lack generalization power (i.e., do not apply to other DDs).

Albinet et al. evaluated the robustness of DPI (Driver Programming Interface) by injecting faults in their functions, thus emulating incorrect usage of these APIs by faulty DDs [Albinet et al., 2004]. Four Linux DDs constituted the case study and the outcomes of the injection experiments were considered from the perspective of kernel responsiveness, feedback and availability (same as Durães and Madeira [2002a]) and, additionally, the workload safety.

Johansson and Suri changed on-the-fly the parameters of kernel functions called by DDs under Windows CE .Net [Johansson and Suri, 2005]. Pre-set values were injected in the called function parameters, in an effort to locate the operational vulnerabilities and quantify their impact on OS robustness. The error model used was later improved in [Johansson et al., 2007b] and the impact of the injection trigger was studied in [Johansson et al., 2007a].

Mendonca and Neves implemented a *SWIFI* technique to test the robustness of DDK APIs for functions used by 95% of Windows XP and Vista DDs [Mendonca and Neves, 2007]. They studied the DDs found in the Windows repository (a folder containing the binaries of the DDs present in the system) for the imported kernel APIs. A number of twenty kernel functions were found to be used by most of the DDs, so they were injected with faults manifesting themselves as malformed function parameters. For each function and each parameter thereof, a dummy DD containing a malformed call of the respective API was built and automatically exercised in order to trigger the injected fault. Using this approach, Mendonca and Neves showed that most kernel APIs are "*unable to completely check their inputs*", indicating a possible robustness issue.

Unfortunately, Mendonca and Neves selected the targeted functions statically, irrespective of (a) the actual usage of the DDs (most of the DDs present

in repository are not loaded by Windows) and (b) the actual usage of the selected functions (some APIs are seldomly used). We believe that a more useful quantification of the OS kernel robustness must consider the actual usage patterns of DDs and kernel functions. Therefore, we provide a more appropriate way to select such functions in Chapter 7, as a contribution of this thesis.

Microsoft provides a powerful robustness testing tool for DDs developed for Windows OSs, the *Device Path Exerciser* (called also *DC2*). In contrast with the previously mentioned approaches which inject fault in DDs to reveal robustness issues in other OS components, *DC2* explicitly targets the robustness of DDs. *DC2* is available to DD developers as part of the Windows Device Testing Framework (WDTF) and Driver Test Manager (DTM), included in the WDK for running test for submission to the Windows Logo Program [Orwick and Smith, 2007, pp. 676]. That is, a DD which does not pass the *DC2*'s robustness tests is not accepted for mass distribution. *DC2* tests at random various error conditions for the supplied DD sending it bursts of (correct and malformed) I/O requests over short time intervals. Thus, not only DD robustness is tested, but also its timing and performance issues are revealed. Microsoft recommends running *DC2* while *Driver Verifier* is also activated, to ease the location of the defects found by *DC2*.

The main issue with the results provided of *DC2* (which are used to decide mass distribution of the respective DD) is its randomness. That is, a DD that passed *DC2* test is not necessarily bug-free one. It only means that *DC2* could not find bugs for the executed tests. This aspect directly relates to the coverage of the respective tests. In Chapter 7 we provide a mechanism to visualize the covered state space of a tested DD, and also show and discuss the areas covered by *DC2* alongside the areas covered by real workloads.

2.3.3 In Isolation

To protect faulty DDs from affecting the rest of the OS, a paradigm assuming the running the DDs in isolation exists. In order to validate individual approaches from this perspective, *SWIFI* was the tool of choice used by many researchers. Examples include tools like *Nooks* [Swift et al., 2002] or *SafeDrive* [Zhou et al., 2006]. Both approaches suppose that the execution of the real DD (or a slightly modified version thereof) inside a virtual execution environment. All execution traces of the targeted DD are monitored to prevent potential failures from propagating to the rest of the OS, but this comes at a high performance overhead.

In 2007 Herder et al. also presented a procedure for detecting and recovering from DD failures [Herder et al., 2007]. However, the focus was on

constructing new OS recovery mechanisms and not necessarily using existing ones. Therefore, as the authors use a low-impact OS (the Minix 3), its application for protecting existing OSs from DD failures is limited. The recovery from DD failures is therefore specific to Minix and hardly applicable to current COTS OSs.

2.4 Operational Profiling as Guidance for Testing

As indicated by the discussion on the state of the art and practice in testing mentioned in Section 2.3, the state space covered by existing testing tools, as well as the selection of test cases determines the adequacy and accuracy of the respective testing method. In other words, in the context of limited test resources, only covering the right states of the DD ensures the failure-free functioning of the respective DD.

Hence, the construction of representative operational profiles is a key step for assessing, and hence improving, the reliability of such complex systems as DDs, with application to testing, performance profiling, and dependability benchmarking. Next, we describe the need of testing driven by the operational profiles as resulted from multiple research efforts, and describe various current uses of the operational profiles.

2.4.1 Why Profiling the Operational Phase?

Results from the area of software defect localization show that faults tend to cluster in the OS code [Chou et al., 2001; Möller and Paulish, 1993]. This effect is explained by the use of large development teams where unequally experienced programmers work on the same code base, thus the distribution of faults is not uniform over the resulting source code [Fenton and Neil, 1999].

Efforts have been made to predict where the faults are in large software systems, employing sophisticated statistical methods ranging from regression mechanisms [Ostrand et al., 2004], failure-proneness models with code churning metrics [Bhat and Nagappan, 2008; Layman et al., 2008], to cluster analysis [Dickinson et al., 2001]. Also, the influence of the organizational structure of the company producing SW was also studied in an effort to indicate the failure prone binaries [Nagappan et al., 2008].

Musa's work on reliability engineering suggests that the overall testing economy can be improved by prioritizing testing activities on specific functionalities with higher impact on the component's runtime operation [Musa, 1993, 1994a, 1996, 2004].

Similarly to Musa, Weyuker also recommends focusing on testing functionalities with high occurrence probabilities [Weyuker, 2003; Weyuker and Jeng, 1991]. Thus, a strategy aimed at clustering the code into functionality-related parts and then testing based on their operational occurrence is desirable, especially when the resources allocated for testing are limited. Weyuker underlines the necessity to test COTS components in their operational environment even though they were tested by their developers or third-party testers [Weyuker, 1998].

Rothermel et al. empirically examined several verification techniques showing that prioritization can substantially improve fault detection [Rothermel et al., 2001]. Specifically, they assert that "*in a testing situation in which the testing time is uncertain (...) such prioritization can increase the likelihood that, whenever the testing process is terminated, testing resources will have been spent more cost effectively in relation to potential fault detection than they might otherwise have been*".

Accordingly, the work presented in this thesis helps focusing testing onto the states of the DD-under-test based on their occurrence and temporal likelihoods to be reached in the field.

2.4.2 Operational Profiles

Under the premise that good reliability estimates are depending on testing as if the SW is executing in the field, John Musa initially developed the concept of *operational profile* while working as the head of the software reliability engineering department at AT&T Bell Labs [Musa, 1992, 1993]. The first operational profiles were built for AT&T Bell projects and are still under active use ever since. Musa compiled most of his research in the seminal book "*Software Reliability Engineering: More Reliable Software Faster and Cheaper*" [Musa, 2004].

The significance of operational profiles for software engineering is apparent in the large volume of research published by two major conferences (IEEE International Symposium on Software Reliability Engineering – ISSRE, and ACM International Conference on Software Engineering – ICSE). At these two conferences, operational profiling still represents a hot topic, even though more than twenty years have passed since they were first introduced in John Musa's work.

As considerable amount of work is dedicated to operational profile research and applications of operational profiles (ISSRE 2008 published 30 research papers and hosted 24 industrial presentations), in the following sections we summarize the most important research projects, grouped by their main application area.

In Software Engineering

According to Musa, a *profile* is "*simply a set of disjoint (only one can occur at a time) alternatives with the probability that each will occur*" [Musa, 1993]. Key to obtaining valid and accurate profiles is the usage information, and Musa believes that this information is either readily available or it can be estimated.

Using the SW system's usage information, engineers derive operational profiles, by manually quantifying the expected usage of each element in the system. The recommended approach to obtain operational profiles is a multi-stage process from user profiles to operation. At each step (customer, user, system mode, functional) a profile is built. The process concludes with the construction of an operational profile that encapsulates all the previous profiles. For instance, an operational profile of an OS includes information about the user profile (programmer, database administrator, gamer), system mode (the typical workload used by the selected user type), etc. [Musa, 2004].

Once obtained, an operational profile can be used for various purposes. For instance, Musa also introduced *SRET* (*Software Reliability Engineered Testing*), a concept that uses the operational profiles for test case selection [Musa, 1996].

Musa used operational profiles to study the effects of errors to field failure intensity [Musa, 1994a] and later suggested improvements based on the observed results [Musa, 1994b].

In an recent article Hassan et al. showed that the operational profiles obtained based on estimations of the field conditions usually differ from the actual usage, so the profiling should be done (where possible) based on real field data (log files, other mechanisms that capture runtime behavior) [Hassan et al., 2008].

In Logging Systems

While obtaining actual field data is not viable in many situations (i.e., user privacy issues, additional performance and resource overhead of logging), often logging systems are used to obtain better estimates of the SW system's field operation. Sometimes, beta-testing[5] is also used to collect field data.

When available, logs are analyzed and operational profiles are constructed from the usage information contained in the logs. The benefit from operational profiles is two-fold: (a) in *debugging* by providing support for locating

[5] *Beta-testing* uses a pre-release version of the product for external testing purposes, in order to identify configurations that cause problems, as well as collect requirements and suggestions from users.

the defect that caused a system failure and (b) in *performance improvement* by identifying and releasing pressure from the highlighted execution bottlenecks (execution hotspots).

Some SW products use logging and profiling systems. Examples include the *Eclipse Test and Performance Tools Platform* (*TPTP*)[Eclipse Project, 2009] or the *Java Virtual Machine Profiler Interface* (*JVMPI*) [Sun Microsystems, 2009a], replaced by the *Java Virtual Machine Tool Interface* (*JVMTI*) in Java 1.5 [Sun Microsystems, 2009b]. Another prominent profiling system is used by Microsoft to collect information about OS crashes (initially *Watson*, later *CEIP – Customer Experience Improvement Program*) [Orgovan, 2008]. Microsoft uses the collected crash information to improve the OS components that caused the failure or to inform the developers of the respective OS components or SW applications, in an effort to improve user's experience with OSs belonging to the Windows family.

Despite sustained efforts to obtain accurate field data, Weyuker and Avritzer acknowledge that constructing operational profiles based on actual usage is difficult [Weyuker and Avritzer, 2002]. Nevertheless, Weyuker and Avritzer hint that code profiling and trace analysis might help considerably towards achieving this task.

Other Uses of Operational Profiles

McMaster and Memon introduced a statistical method for test effort reduction based on the call stacks recorded at runtime [McMaster and Memon, 2005]. While their approach effectively captures the dynamic program behavior, it is specific to single-threaded, user-space programs, though not directly applicable to DDs (as they run in an arbitrary thread context). Leon and Podgurski evaluated diverse techniques for test-case filtering using cluster analysis on large populations of program profiles, highlighting them as a prerequisite for test effort reduction [Leon and Podgurski, 2003].

Avritzer and Larson proposed an approach to describe the load of a large telecom system [Avritzer and Larson, 1993]. They introduce a load testing technique called "Deterministic Markov State Testing" for describing the operational model of a telecommunication system. The incoming and completion of five types of telephone calls define the state of the system. However, as the work was driven by the necessity to test the given telecommunication system, knowledge of the system internals was intensively used thus limiting the applicability of the method to other systems. In contrast to their method, the approach presented in this thesis is generalized to OS add-on components and to DDs in particular.

For Windows WDF DDs, two main tracing techniques exist: (a) the *WPP*

– Windows Trace Preprocessor and (b) the *driver error logs*) [Orwick and Smith, 2007, Chap. 21]. WPP is a kernel-level trace logging facility used to determine the location and context of bugs present in the DD code. The *driver error logs* collect data related to different DD activities, such as setup and update operations and system event logs.

For the approach presented in this thesis, DD activity tracing has value only as an alternative source of obtaining information about a DD's runtime behavior. It requires manual source code-level instrumentation (via WPP) or manual log enabling (driver error logs), while our approach assumes no access to DD's code.

Johansson et al. used the concept of "*call blocks*" to find the best time instant to inject a fault into Windows CE. Net DDs [Johansson et al., 2007a]. Call blocks were obtained on the fly by monitoring the communication interface of the selected DD with the rest of the OS kernel. They could be also considered a form of an operational profile, as they represent runtime behavior in terms of sequences of calls to driver-external functions. Instead of using the more common approach of triggering an error injection on the first call of an external function, Johansson et al. explore the effects of triggering injections on a call block basis. The operational profiling methodology presented in the Chapter 7 of this thesis is similar in the sense that operational profiles are built using an equivalent source of information. Though, the approach presented in Chapter 7 is developed to reveal execution hotspots in DDs in terms of followed code paths.

2.4.3 Code-path Profiling and Trace Analysis

Ball and Larus acknowledged the application of path profiling for test coverage assessment, "*by profiling a program and reporting unexecuted statements or control flow*" [Ball and Larus, 1996]. They used binary instrumentation to obtain instruction traces that revealed a program's control-flow, to identify paths and their execution frequencies. The paths end at loop and procedure boundaries.

An extension is represented by the "*whole program paths*" described in [Larus, 1999], which crosses both boundaries to provide a more accurate picture of a program's execution. Though, these approaches are not directly applicable to DD as the they are implemented as libraries of functions rather than programs in the classical sense. Moreover, instrumentation induces a high execution overhead and produces large amounts of data, two characteristics which penalize the use of this approach inside the OS kernel space.

Leon and Podgurski used profiles generated by individual test cases and a clustering technique for evaluating test suite minimization by selecting one

test case per cluster [Leon and Podgurski, 2003]. The profiles used were generated by third-party tools, so the cluster analysis had to rely on their accuracy.

While test cost reduction is outside of the scope of this thesis, we focus on building viable and accurate DD profiles, as a prerequisite to reducing test efforts. Further described in Chapter 7, our methodology creates such profiles for kernel-mode DDs by revealing the code paths taken and the set of driver-external functionalities required at runtime. By profiling a DD's activity, the work presented in this thesis guides a rigorous partitioning of the code by indicating runtime execution hotspots. As our methodology is disconnected from the need to access any OS part's source code, it can be used for black-box level DD profiling, to ease the testing campaigns targeting DDs.

2.5 Chapter Summary

This chapter presented the context of the OS testing problem that this thesis intends to solve, alongside with a survey of the state of the art and of the practice in the field of verification and validation (V&V). On this basis we first identified the main causes of OS outages. In their role of mediators between the OS and the hardware parts of a computing system, DDs have been found to be the main source of OS failures. After the functioning of most common V&V techniques has been presented, the discussion continued with the presentation of current verification tools specifically designed for OS components and DDs. For each tool, the benefits and disadvantages in terms of test space coverage and test case selection were discussed, as these aspects explain why DD still continue to be faulty despite sustained test efforts. Further, operational profiling was presented as a possibly viable solution to guide testing onto the DD states having a higher likelihood to be reached in the field, and several profiling approaches were introduced.

Chapter 3

System Model and Driver State Model

What is the device driver's role in current OSs and how can its activity be modeled? How can the state of a device driver be modeled in a useful way for guiding testing?

Given the continual increase in code size and complexity of OSs (and implicitly, of DDs), we believe that a proper characterization of the area targeted by tests is a prerequisite of primary importance for improving current testing paradigms.

Under the assumption that the source code of the tested DD is not available to testing, we present a model of the state space of a DD inferred only by utilizing the communication traffic available at the DD's interfaces to the rest of the OS. The DD model and the methodology to infer its state space presented this chapter constitute one of the key contributions of this thesis, namely **C1** (see Section 1.2.2).

Consequently, this chapter first presents and discusses the architecture of DDs in the most common two COTS OSs, Microsoft Windows and UNIX/Linux. Then, the system model is developed and discussed from the perspective of the SW and hardware components directly communicating with DDs. Subsequently, various DD communication interfaces are investigated as primary information sources revealing the runtime activity of the DD. Using the I/O traffic, an abstraction of the DD's state is developed – the *driver mode*. Finally, this chapter concludes with building a DD's complete state space as a state-diagram containing all possible modes that the respective DD can be in at any chosen time instant.

3.1 Device Drivers in Current COTS OSs

Device drivers (DD) are SW components that provide to user applications (and to OS) a homogeneous programming interface to diverse hardware devices. DDs are usually implemented as add-on components that can be added to a compatible OS by system administrators (users having administrative privileges for the respective OS installation).

In this section, we present and compare both Windows and Linux DDs from architectural and internal routines perspectives. We have chosen these two OS families as they are currently the most popular COTS OSs, each with tens of millions of installations in the world. In this thesis we chiefly use Windows DDs as case studies for the introduced concepts and methods though these are directly portable to Linux DDs (and other OSs), due to similar architecture of the DDs belonging to the two classes of OSs. To substantiate this, the section concludes by discussing the similarities of the two families of DDs.

3.1.1 Device Driver Architectures

When speaking about the architecture of the Windows and Linux components, the historical connection of the two OSs is useful. Both of these OSs have the same ancestor, namely the UNIX OS. Linux was created as a UNIX clone by Linus Torvalds in 1991. Linux refers hardware devices using a number (a combination of minor and major numbers) and DDs are considered files. In current Linux versions three types of devices exist: *character devices* (accessed sequentially without buffering), *block devices* (accessed randomly, data accessed in blocks) and *network devices* (accessed using socket API).

The history of Microsoft OSs starts much earlier. In 1979 AT&T decided to sell UNIX as a commercial product and Microsoft purchased the mass distribution license for UNIX V7 (seventh edition). One year later, Microsoft licensed XENIX, a new OS for PDP-11 machines. XENIX was based on UNIX V7 and BSD 4.1, containing also Microsoft enhancements (i.e., multiple virtual consoles – later inherited by Linux). In 1981 MS DOS v1.0 was released, having a similar driver architecture to XENIX, the main difference being the fact that the DOS DDs were built-in and they were not considered files anymore (as in Linux OSs). MS DOS v2.0 introduced loadable DDs, thus supporting the development of new DDs by the hardware device producers. Windows 3.1 utilized an DD architecture based on MS DOS. It was the Windows 95, 98 and NT that made an important step forward, by introducing the WDM (Windows Driver Model) [Dekker and Newcomer, 1999; Oney, 2003].

WDM appeared as an effort to create a driver architecture which is *forward-compatible* (all next Windows family OSs use WDM). WDM-compliant DDs are usable on all of Microsoft's recent OSs (from Windows 95 to Vista SP1). WDF (Windows Driver Foundation) was later introduced as an wrapper for WDM to ease the development of Windows kernel DDs and enable user-mode DDs. Vista and even the newest Windows 7 (at the time of writing this thesis still a beta-version) use WDM/WDF DDs [Microsoft Corp., 2009c].

In the following, we detail the architecture and functioning of both Windows and Linux DDs, concluding with a discussion on the observed similarities. Such a discussion constitutes the key basis for understanding the choices made to build the system and DD models (Section 3.2) and the DD state model (Section 3.3).

WDM/WDF Driver Architecture

To illustrate how an I/O request is processed by a DD in Windows OSs, consider a simple example of an application that issues a `read` request to a hardware device, for instance the hard-disk drive. Figure 3.1 depicts this procedure where the main stages are: (1) the application calls the `ReadFile` function of the system calls library Win32 API; (2) the Win32 API traps the OS kernel into the I/O Manager which selects the DD managing the hard-disk drive; (3) the I/O Manager encodes the I/O request in an "*I/O request packet*" structure (presented and detailed soon) and forwards it to the hard disk DD; (4) the DD contacts the HAL (Hardware Abstraction Layer[1]) which, in turn, retrieves the actual data and completes the I/O request packet; (5) the I/O Manager reads the completion information from the I/O request packet and (6) returns the result to the Win32 API, in terms of a pointer to read data; (7) the Win32 API copies the data to a buffer accessible to the calling application (in user space) and (8) informs the application about the result of the operation and the location of the requested file.

The steps (3') and (4') are actually composed of multiple stages. As in Windows DDs are internally organized in stacks, at step (3') the initial I/O request packet is sent (and modified along the way) to the driver located below the current one, until the HAL is reached. At step (4') each driver completes the I/O request packet coming from HAL, eventually sending it back to the I/O Manager when the topmost driver is reached, at step (5).

[1]HAL is a SW layer that deals directly with the hardware. HAL's purpose is masking all hardware-specific information regarding the characteristic of the actual hardware device in order to ensure an unified access interface [Microsoft Corp., 2006].

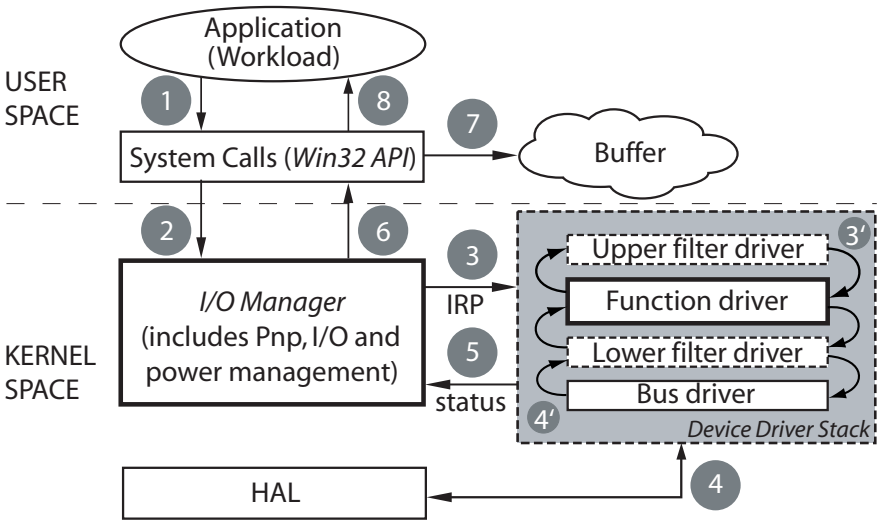

Figure 3.1: The WDM/WDF driver architecture.

WDM describes several different driver types [Oney, 2003]. WDM uses a layered architecture of DDs similar to the one depicted in Figure 3.2. Each hardware device has at least two associated DDs, the *function driver* and the *bus driver*. The former handles proper managing of the associated hardware peripheral, while the latter manages the communication bus which connects the respective peripheral to the rest of the computing system (i.e., ISA, PCI, USB etc.).

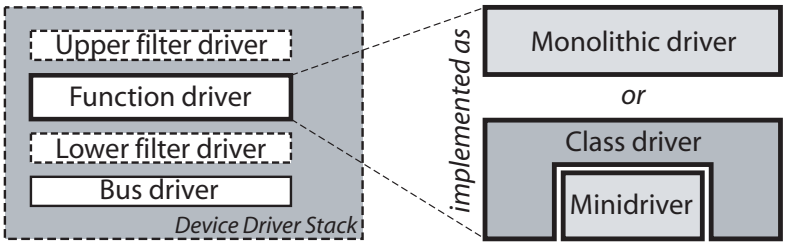

Figure 3.2: The WDM/WDF driver implementation. The *function driver* is implemented either as a *monolithic driver* or as a combination of a *class driver* (provided by the OS) and a *minidriver*.

Some devices have additional driver layers, consisting of *filter drivers* which wrap the function driver. A filter driver is responsible for modifying the behavior of the main function driver and it can be located either above or below it. WDM does not limit the number of filter drivers a hardware device can have. This mechanism permits incremental changes to the main behav-

ior of the function driver, thus enabling modifications that would otherwise require source code access to the function driver.

Additionally, the function drivers come in two flavors, depending on their implementation: as *monolithic* and as combinations of a *class driver* and a *minidriver*. A monolithic driver encapsulates all of the functionality needed to support a hardware device. The same functionality can also be modularly implemented as a combination of a class- and a minidriver. In this case, the class driver manages the entire generic class of devices, while the minidriver handles only the vendor-specific functional characteristics of a certain device. Usually, the class drivers are provided by Microsoft and writing the minidriver is generally the hardware manufacturer's task.

I/O Request Packets

The *I/O Request Packet* (termed henceforth IRP) is an OS kernel structure built by the I/O Manager when a request needs to be sent to a DD. The IRP structure contains the request type and the parameters needed by the recipient DD to start executing the request-associated activity. When a result of the operation is available, the DD uses the same IRP structure to piggyback it back to the I/O Manager.

Currently, WDM/WDF specifies 28 types of IRP requests (for instance READ for reading data from the device and CLEANUP for preparing the device for unload etc.). A DD must implement dispatch functions for every IRP type it supports and register its list of supported IRPs with the I/O Manager. This request type-based code separation of WDM-compliant DDs is relevant for our approach, as one can infer the functionality executed at any instant by a DD, based only on the type of the received and completed IRPs.

Linux Module Architecture

The DD architecture used in Linux OSs is similar to the WDM architecture used by the latest Windows OSs. When visually compared with the Windows architecture (see Figure 3.1 versus Figure 3.3), this similarity becomes apparent. In Linux, the role of POSIX[2] is the same as the role of the Win32 API system call libraries in Windows.

In Linux there is no clearly defined entity mediating the I/O, PnP and Power issues (the I/O Manager in Windows), and there is no clear distinction

[2]POSIX are UNIX-wide specifications for system calls, each UNIX/Linux clone using an own implementation of the POSIX specifications.

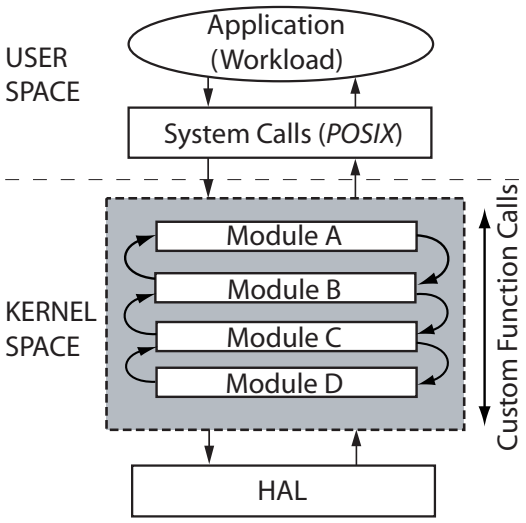

Figure 3.3: The Linux module architecture.

between the layered *modules*[3] as the bus, functional or filter drivers in Windows. Also, while Windows uses a set of well-defined IRPs to communicate with DDs, in Linux the modules communicate with each other using function calls with custom parameters instead.

Current versions of Linux use a HAL layer whose implementation depends on the actual hardware platform of the machine that the Linux kernel is compiled for, similar to Windows.

3.1.2 Device Driver Routines

Both Windows and Linux DDs consist internally of various routines. A subset of them are mandatory for the OS kernel to be able to communicate properly with the respective DDs and some are optional depending on the actual DD and hardware device type, operations etc. Without going into the details of these DD routines, we briefly present the most relevant ones for the approach presented in this thesis. For a more detailed description of the DD architectures, several books exist. For Windows see Dekker and Newcomer [1999]; Oney [2003] and Orwick and Smith [2007], while for Linux a good reference is Corbet et al. [2005].

[3]In UNIX/Linux the DDs are commonly called "*modules*".

Windows Driver Routines

A loaded and active Windows DD is represented by a special OS structure, termed as *DriverObject*. The *DriverObject* contains pointers to different routines that the associated DD can perform.

Maintenance routines. When the DD is loaded into the OS kernel (when the OS boots or when a new PnP hardware device is added to the running computer), the *DriverEntry* routine fills the *DriverObject* OS structure with pointers to the kernel memory addresses where different I/O-handling routines of the DD are located. By knowing the location of those routines, the OS kernel is enabled to call directly each necessary DD routine. Similarly, when the DD is unloaded from the system, the *DriverUnload* routine removes these pointers and destroys the *DriverObject*.

Once the *DriverObject* structure is filled with the necessary pointers, the DD needs to be initialized and bound to a hardware device. This is the responsibility of the *AddDevice* routine. The *AddDevice* allocates the necessary resources, creates and populates another kernel structure representing the hardware device itself – the *DeviceObject*. This structure holds the name associated to the hardware device which, in Windows, is a global unique identifier (GUID). Once a device object is created, the respective hardware device can be referred to using a handler obtained by providing its GUID. To avoid naming ambiguities, Microsoft provides a tool in the DDK for automatically generating GUIDs [Oney, 2003].

Dispatch routines. Another important class of routines present in WDM/WDF-compliant Windows DDs are the so called *dispatch routines*. They are responsible for handling incoming I/O requests packed as IRPs. Based on the IRP type, a corresponding dispatch routine is selected and executed. The I/O Manager selects the right dispatch routine using the information published at DD initialization in the *DriverObject* structure. Every IRP structure have fields containing parameters required by the respective I/O dispatch function. When a dispatch routine receives an IRP, it reads the operation's parameters and starts processing it by either sending the work directly to HAL (if the respective driver is located closest to HAL in the driver stack – see Figure 3.1) or to the driver located immediately below it in the stack.

To ensure a minimal compatibility between the Windows kernel and DDs, WDM/WDF-compliant DDs must implement the dispatch routines listed in Table 3.1, the rest of the dispatch routines are optional.

For brevity, henceforth we use only the actual name of the I/O operation when referring to a IRP (that is, without the *IRP_MJ_* prefix).

Table 3.1: Dispatch routines required in WDM/WDF-compliant DDs [Oney, 2003]

Full IRP Name	**The dispatch routine manages...**
IRP_MJ_CREATE	the creation of *DeviceObject*
IRP_MJ_CLEANUP	the destroying the driver-created structures
IRP_MJ_CLOSE	the actual removal of the DD from the system
IRP_MJ_READ	*read* requests to the hardware device
IRP_MJ_WRITE	*write* requests to the hardware device
IRP_MJ_PNP	Plug-and-Play activities
IRP_MJ_POWER	Power activities (stand-by, hibernate, power off)
IRP_MJ_DEVICE_CONTROL	I/O control requests to a hardware device
IRP_MJ_INTERNAL_DEVICE_CONTROL	I/O control requests specific to a certain device
IRP_MJ_SYSTEM_CONTROL	Windows Management Instrumentation (WMI)

Linux Module Routines

Linux modules are internally similar to Windows DDs as they also contain different specialized routines to control I/O operations and the hardware devices. The initialization and unload routines of a Linux module do not bear predefined names as in Windows. Instead, module developers can choose custom names for those routines and register them with the Linux kernel by using the *module_init* and *module_exit* macros.

Maintenance routines. The routine designated as module initialization routine in the *module_init* macro must specify the DD's name, a set of routines for file operations (the Linux counterpart of Windows dispatch routines) and a major number for the hardware device associated with the respective DD. In Linux, each hardware device is associated with a number ranging from 1 to 255 (called device *major number*). In addition, each DD handling one of the 256 devices can manage as many as 256 devices (DD-managed devices are designated by a *minor number*). Hence, in Linux up to 65535 devices can be accessed, as major number 0 is reserved for automatic device assignment. On DD unload, the routine designated by the *module_exit* macro de-registers all the associations made by the initialization routine.

File operations. In Linux the *file operations* handle all I/O activity of the module. In Linux almost everything is considered to be a file, even external hardware devices are accessed as regular files after they are mounted to the local file-system, thus the name for Linux I/O operations. Typical modules implement the following file operations (names are self-explanatory): *Open*, *Read*, *Write*, *Seek* and *Close*. For handling device-specific parameters, the *IoCtl* operation is also implemented by certain Linux modules.

3.1.3 Comparing Windows and Linux Drivers

To warrant the generality of the driver model subsequently built in the next sections of this chapter for current COTS OSs, we now briefly summarize the main similarities and differences between Windows and Linux DDs.

From the *architectural perspective* the DDs in the two OSs are very similar to each other. The I/O processing mechanism is more complex in Windows, Linux does not have an OS entity centralizing I/O, PnP and Power activities (the Windows I/O Manager). Both OSs use a system call layer and an HAL layer, even though they are implemented differently. In Windows DDs are stacked and different DDs communicate to each other in a well-specified order and fashion, using IRP structures. In Linux the inter-module communication is open, developers deciding the details based on the purpose of the module. Messages are passed from module to module (or to HAL) using customizable function calls. Hence, we consider that the Windows driver architecture is more restrictive then Linux's from this perspective.

From the *internal routines perspective* the two OSs are very similar to each other. Both use routines to register and de-register DDs with the OS kernel, and both use dispatch routines specialized for handling certain I/O operation types. As a note, Windows internal structure is more regulated than in Linux, this reducing the possible incompatibilities between DDs written by different developers (see Windows' fixed number of IRPs, device naming using unique GUIDs, etc.).

3.2 System and Device Driver Models

In this section we develop the *system model* and subsequently, the *driver model* used throughout this thesis. Given the commonality between the two current families of COTS OSs shown in the previous section (Windows and UNIX/Linux), we believe that our model is representative for systems equipped with any of the two OSs. Moreover, our system model is abstract enough to represent most computing systems equipped with monolithic OSs.

3.2.1 Involved OS Structures and Components

We now introduce our system model and the background behind the state model for DDs. Figure 3.4 represents a typical computer system equipped with a COTS OS supporting a set of applications (the *system workload*) using services provided by the OS.

This thesis focuses on the communication interfaces between the *I/O Manager* and the *DDs* located within the OS kernel space. The *I/O Manager*

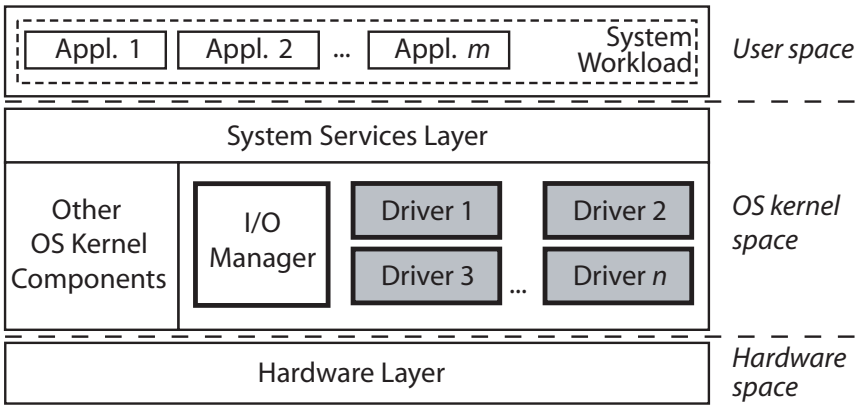

Figure 3.4: The considered system model, representing a general-purpose computing system equipped with an OS mediating between user applications and hardware peripherals.

is a collection of OS structures responsible for mediating the flow of I/O, PnP and Power requests between the applications and the DDs. A *DD* is an independent OS component handling the communication with one or more equivalent peripherals. In our model DDs act on I/O requests initiated by the I/O Manager directly or on behalf of user applications.

Our system model does not differentiate between a single DD placed on the I/O path between the OS kernel and the hardware and driver stacks. This abstraction enables our model to be used independently of the actual DD architecture of a specific OS. Also, in our model the I/O Manager is an abstraction of the Windows I/O Manager presented in Figure 3.2, in the sense that it only represents the other communication end from the DDs perspective. That is, in our model the I/O Manager has only two roles: (I) it constructs and serializes the I/O requests to be sent to the DD, and (II) constitutes the recipient of the I/O operations results provided by the DD.

While our approach is applicable for generic DDs and OSs, we utilize Windows XP and Vista DDs as representative case studies for the proposed concepts in the following chapters of this thesis. Hence, onwards we use the terms established by Windows to refer to DDs and related aspects.

3.2.2 A Driver's Communication Interfaces

The communication flow between the I/O Manager and the DDs has a key importance for the DD profiling methodology presented in this thesis, as this flow is analyzed to characterize the activity of a DD. At this level Windows uses different communication schemes as specified by the Windows Driver

Model (WDM) and described in the following sections [Dekker and Newcomer, 1999; Oney, 2003].

I/O Call Interface

In Windows, the I/O Manager builds a request data structure and populates it with the parameters necessary for the DD to start resolving the request. Next, the I/O Manager informs the responsible DD that a request is available. When the DD finishes executing the code associated with resolving the request, it fills the result fields of the structure and passes it back to the I/O Manager. The I/O Manager unpacks the data structure and forwards the results to the user space application that requested the I/O. The data structures used for passing the request between the I/O Manager and DDs are the IRPs.

In Linux, applications make direct calls to a specific DD by first obtaining a handle to it by using its major and minor numbers. Once the handle is obtained, the user application issues the actual I/O request to the selected DD. The DD performs the associated activity on the hardware device and returns the result in a kernel buffer. Next, the content of the kernel buffer is copied in an user-space buffer supplied by the calling application.

In both Windows and Linux each incoming I/O request triggers execution of a certain dispatch routine. At this interface (onwards termed "*I/O call interface*") we monitor the I/O traffic and analyze it to gain insight on which routines are executed at any time instant.

Functional Interface

Often DDs are implemented as *dynamic-linked libraries* (DLLs), which are libraries of functions. The functions implemented in DDs get executed when they are called by the OS (directly, or on behalf on user applications). Like any other DLL library, a DD might call functions which are implemented in other DLL libraries. As most current DDs execute in kernel space, they can only link at runtime to libraries of functions also located in kernel space.

At runtime, a DD calls functions located in DD-external DLLs as required by the current activity performed by the respective DD. Therefore, the DD's evolution in time can be characterized by monitoring this interface (onwards termed "*functional interface*"). In Chapter 7 we use this communication interface in addition to the I/O call interface to infer the followed code paths taken inside the DD at runtime.

Hardware Interface

Some of the hardware devices that attach to a communication bus (ISA, PCI, PCI-Express etc.) generate interrupts to signal the OS that they need service [Orwick and Smith, 2007]. Hence, the function driver for such a device must include code to manage interrupts when they occur in the operational phase.

Hardware devices generate two types of interrupts, *line-based* and *message-based* interrupts. Older devices generate line-based interrupts, which are electrical signals on a dedicated interrupt line. Newer devices generate message-based interrupts, which represents data written to a special memory address. Prior to Vista, only line-based interrupts were supported in Windows.

A DD that manages interrupts has to implement two classes of routines: the *Interrupt Service Routines* (ISR) and the Deferred Procedure Calls (DPC). When a hardware device interrupts, the OS calls the associated DD to handle the interrupt. The respective DD then starts executing its ISR, which is responsible to (a) check if the associated device is interrupting; if yes, then (b) stops the device from interrupting and saves all device context and, finally, (c) it calls the DPC routine and returns. The DPC performs the actual device-specific interrupt handling and, after finishing, it re-enables the interrupts for the respective hardware device.

Devices that attach to USB, FireWire, Bluetooth and other protocol buses do not generate interrupts, so their DDs do not contain ISR and DPC routines. In this thesis, we only consider the I/O call and the functional interfaces of a DD for developing the framework to model its operational behavior.

3.3 Driver State Model

From a testing perspective, the ability to precisely pinpoint which functionality the system-under-test executes at any specific instant is of critical importance as a basis for observability. This is not a trivial task when testing OSs (or components thereof) as they are complex and dynamic, entailing a high level of non-determinism and often being delivered without their source code. These constraints constantly challenge the software testing community to investigate new methods to define and accurately capture the *state* of such a system.

In this thesis we consider that the source code of the DD is not available. Except the situation when testers indeed have access to the DD's source code, this assumption represents the regular circumstances for testing add-on, COTS DDs. If the source code becomes available, the validity and value

of our approach is maintained and can be enriched with relevant information obtained from the source code, as needed.

Therefore, we consider the DD state being characterized by the handled I/O requests. As we assume no access to the DD's source code, we are constrained to use a relaxed definition of *state* to accommodate only the available information (i.e., the observable communication at the interfaces of the DD with the OS kernel). Onwards, we define this relaxed state as the "driver *mode*".

3.3.1 Driver Mode and Transitions Between Modes

A DD is idle from the initialization instant until the first IRP request is received. The DD is in the "*processing* IRP_i" state from the instant when IRP_i is received and until the DD announces its completion. In each mode, the DD executes a dispatch routine determined by the type of the received IRP, as specified by the WDM [Oney, 2003]. This situation is depicted in Figure 3.5.

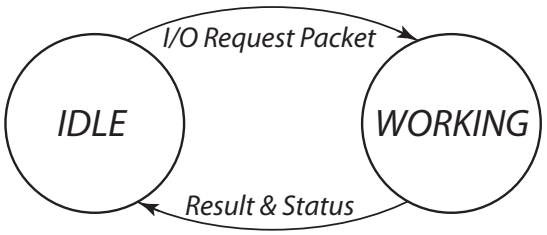

Figure 3.5: The basic *idle*↔*working* functioning cycle of a DD. Initially the DD is *idle*, incoming I/O request trigger state change to *working*. When the processing of the initial request is finished, the DD outputs operation status and returns to the *idle* state.

Actually, some I/O requests are processed concurrently by the DD, i.e., processing IRP_j can start before IRP_i is completed (assuming that IRP_i was initiated before IRP_j but its activity is not finished yet). To the current extent of our experimental work we have never encountered situations when more than one I/O requests of the same type are processed at once. Hence, we define the *mode* of a driver D as follows:

Definition 1 (**Driver Mode**). *The mode of a driver D is a n-tuple of predicates, each assigned to one of the n distinct I/O request types supported by the driver:*

$$M^D : \quad < P_{IRP_1}\ P_{IRP_2}\ ..\ P_{IRP_i}\ ..\ P_{IRP_n} >,\ where$$

$$P_{IRP_i} = \begin{cases} \textbf{1}, \text{ if D is currently } \textbf{performing} \text{ the functionality triggered by} \\ \quad \text{the receipt of } IRP_i \\ \textbf{0}, \text{ otherwise} \end{cases}$$

From this definition it results that the driver mode is a binary tuple of size n, and therefore the total state space size (the total number of modes) for a driver D is 2^n.

Definition 2 (**Transition**). *A transition between two driver modes is an instantaneous event triggered by receiving or completing I/O requests.*

As the I/O Manager serializes both the sending and receival of I/O requests, only one bit can change at a time in the n-tuple describing the driver mode. Thus, a DD can only switch to modes whose binary tuples are within Hamming distance[4] of 1 from the current mode. As the mode is a tuple of length n, there are n transitions possible from each mode, implying the total number of transitions in our model to be $n \cdot 2^n$.

A key observation for further developing of the concepts presented in this thesis is that the driver modes are associated with execution of code, and therefore, have computational duration. That is, while transitions are instantaneous, the DD spends a certain amount of time in each visited mode.

To illustrate the concepts of DD modes and transitions between modes let us consider a simple example. Figure 3.6 depicts the temporal evolution of a hypothetical DD supporting four distinct I/O requests, i.e., CREATE, READ, WRITE and CLOSE, in an arbitrary time interval. Consequently, the DD modes are represented by 4-tuples (CREATE, READ, WRITE,CLOSE).

The leftmost bit is set while the DD performs the functionality associated with CREATE (the dispatch routine handling CREATE requests), the second leftmost bit is set while the DD performs READ, the third bit when performing WRITE and the rightmost bit when performing CLOSE. Note that the DD can execute several activities of different types concurrently, in which case the binary string defining the current mode contains several bits set.

In Figure 3.6 the considered DD receives I/O requests (the black triangles on the time axis) and, after a while, finish their execution (the white triangles). The horizontal arrows represents the activity being performed by the DD in the respective time interval (*idling*, *reading*, *writing*), while the vertical arrows represent mode switches (transitions).

[4]The *Hamming distance* between two strings is the number of positions for which the corresponding symbols are different [Hamming, 1950].

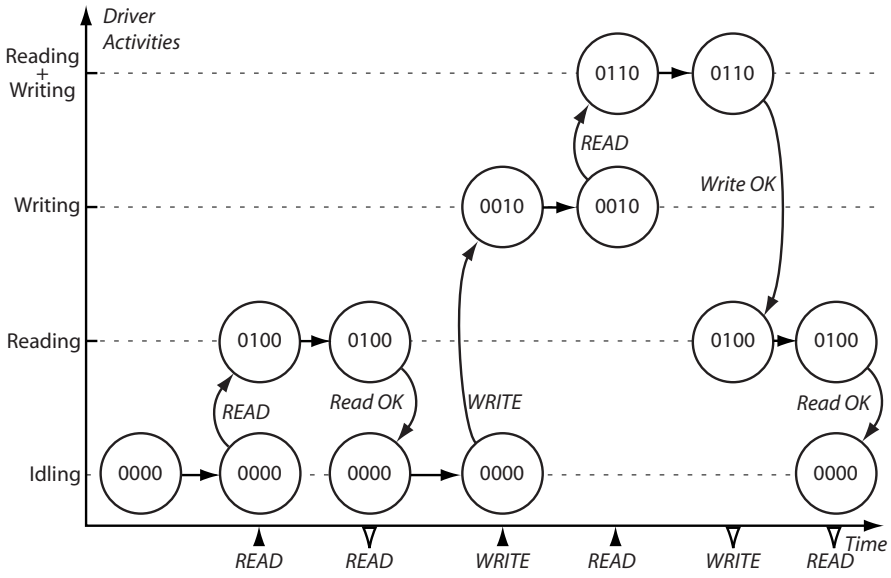

Figure 3.6: The temporal evolution of a DD supporting four I/O requests.

Initially, the considered DD is *idle*, waiting for I/O requests (the DD is in mode 0000). After a while, the DD receives a READ request, so it switches instantly to mode 0100. After some time the DD finishes reading (the READ dispatch routine returns) and the DD switches back to the idle mode. Next, the considered DD receives a WRITE request and, before the WRITE dispatch routine returns, another READ request is received. Hence, according to our definition of the driver model, the DD is considered to be in mode 0110 (both READ and WRITE were received and none of the respective dispatch routines had returned). After a while, WRITE dispatch routine returns, followed shortly by a READ routine return, such that the DD switches back to the *idle* mode.

3.3.2 The Total State Space of a Device Driver

Figure 3.6 depicts the temporal evolution of a DD supporting four I/O request types (CREATE, READ, WRITE and CLOSE), but only for a short time interval and only for few possible sequences of incoming and outgoing requests. As previously mentioned, the complete state space of a DD has to be identified for ensuring accurate and adequate testing, so all possible combinations of I/O requests simultaneously handled by the DD have to be accounted. Consequently, Figure 3.7 represents the integral state space reachable by the considered DD (that is, the complete area that should be covered by exhaustive testing – all modes, all transitions).

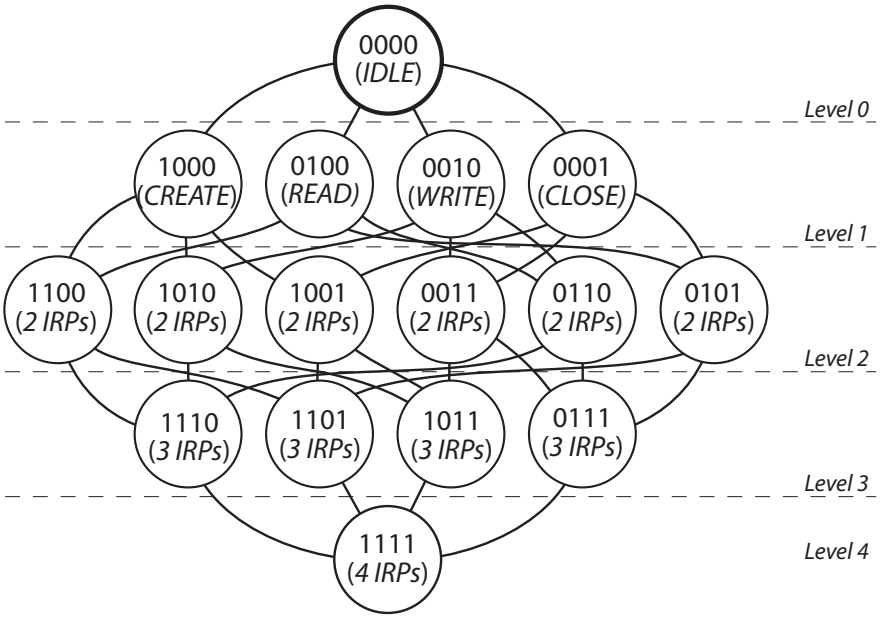

Figure 3.7: The *total state space* of a DD supporting four IRPs. Circles are driver *modes*, lines represent bi-directional *transitions* between pairs of driver modes.

In Figure 3.7 we organized the reachable driver modes on levels. On each level i there are i I/O requests serviced simultaneously by the DD. That is, on level 0 the DD is idling, on level 1 there are four modes each associated with a single I/O request in execution, level 2 contains all combinations of modes where two I/O requests are simultaneously in execution and so forth. For simplicity, in Figure 3.7 the lines are actually bi-directional transitions. As defined, transitions represent one-bit changes in the mode tuple, so transitions are not allowed to jump over neighboring levels.

Generalizing, for a DD supporting n I/O request types, such a representation of the state space has $n+1$ levels, the levels labeled from 1 to n contain modes in which the DD is executing some I/O-related functionality, while level 0 is reserved for the *idle* mode.

Definition 3 (**Total State Space**). *We define a driver's total state space as a digraph with the complete set of modes* $M = \{M_1^D, M_2^D, \ldots\}$ *as vertices and the set of transitions* $T = \{t_1, t_2, \ldots\}$ *as edges. Each transition from* T *maps to an ordered pair of vertices* (M_i^D, M_j^D)*, with* $M_i^D, M_j^D \in M$*,* $i \neq j$ *and the modes* M_i^D *and* M_j^D *within a Hamming distance of 1 from each other [Hamming, 1950].*

As a direct consequence, the total state space size of a driver supporting

n I/O requests contains 2^n modes interconnected by $n \cdot 2^n$ transitions. At this point, we speculate that only few driver modes and transitions are actually visited and, respectively, traversed in the operational phase. This assertion is based on the intuition that not all the theoretically reachable modes are actually allowed by the internal structure of the DD. For instance, is unlikely that CREATE and CLOSE can be processed simultaneously by a DD, as their actions are mutually exclusive. However, the aspects related to the reachable subset of total (theoretically) possible state space of a DD constitute the main subject of the next chapter.

3.4 Chapter Summary

This chapter started by presenting the role of the DDs in the architecture of the currently two most popular COTS OSs families, Microsoft Windows and UNIX/Linux. In this context, we discussed the processing of I/O requests inside the OS kernel and the most important classes of routines present in DDs designed for each of the two OSs. Based on the observed similarities and accenting mainly on the generality aspects, this chapter subsequently developed abstracted system and driver models.

The driver model was used to introduce a relaxed model of a DD's state (that is, the *driver mode*) as a key concept towards identifying the total state space that need be covered by exhaustive testing. As defined in this chapter, the driver mode is based exclusively on the incoming and outgoing I/O requests, thus requiring no source-code level insight for any of the involved OS components. The DD model and the methodology to infer its state space presented in this chapter constitute one of the key contributions of this thesis, namely **C1** (see Section 1.2.2).

Chapter 4

Operational State Space

How can the operational state space of a device driver be determined? How can the test progress be assessed via the operational state space of the respective driver?

Once the total state space of a DD is determined as shown in the previous chapter, a *driver-relevant* workload can be used to exercise the DD for revealing the operational state space of the respective DD – workload combination. After the operational state space is highlighted, the operational profile can be determined, by assigning occurrence probabilities to each of the visited states as suggested by Musa [Musa, 1993]. As the operational state space depends on the workload used to obtain it, the set of states reached in the field can differ from the ones the DD visits in the lab during the test procedures. In an ideal situation, to ensure the adequacy of testing, an operational state space should be obtained in the field and then used to infer the set of DD states that must be primarily covered in the testing process. As this is not always possible, Weyuker suggested that workloads realistically mimicking the field conditions have to be envisioned for adequate testing [Weyuker, 1998].

This chapter introduces an abstract representation of the operational state space of a DD as a key prerequisite for thesis contributions **C2** and **C3** (Section 1.2.2). Test coverage metrics are developed to support assessing the test progress. The importance of highlighting the operational state space *before* starting the actual testing process is also discussed. This chapter concludes by presenting various aspects related to the operational space size versus the total space size by making several hypotheses, used onwards as guidance for an experimental case study – obtaining the operational state space of the default serial port DD for Windows XP SP2.

4.1 The Operational State Space of a Device Driver

To introduce the concept of *operational state space* for DDs, let us reuse the example of the DD supporting four distinct I/O requests (i.e., CREATE, READ, WRITE and CLOSE) introduced in Section 3.3.1, Figure 3.6. The leftmost bit is set while the DD performs the functionality associated with CREATE, the second leftmost bit is set while the DD performs READ and so forth. Note that the DD can execute several I/O activities of different types concurrently, in which case the binary string describing the driver mode contains several set bits.

Figure 3.6 depicted the temporal evolution of the considered DD under the effect of the incoming and outgoing I/O requests generated by a hypothetical workload. Subsequently, Figure 4.1 represents the same temporal evolution, using the total state space representation as depicted in Figure 3.7.

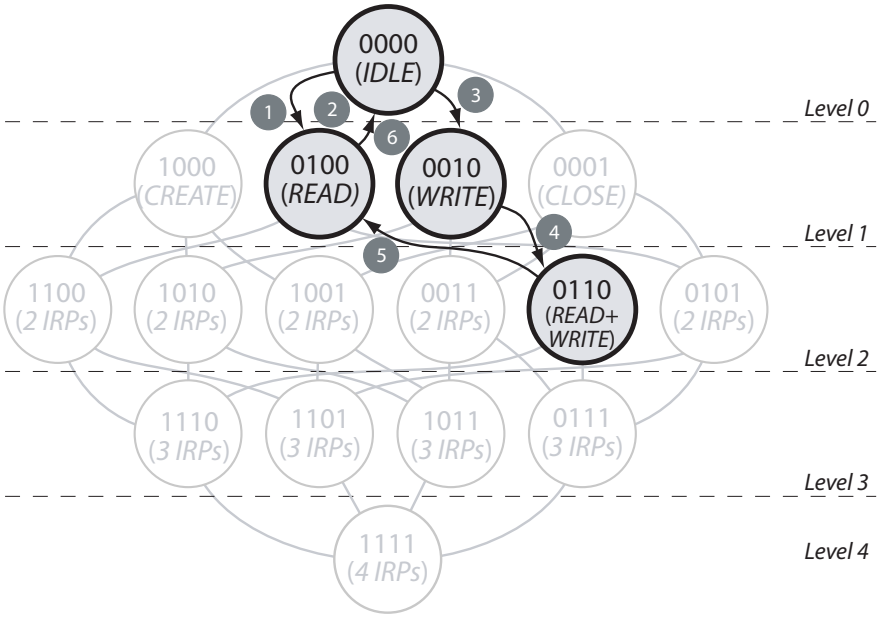

Figure 4.1: The *operational state space* of a DD supporting four IRPs.

In Figure 4.1 the subset of modes visited by the DD is highlighted together with the traversed transitions. The transition labels represent the temporal evolution in a step-by-step fashion. That is, at step 1 the DD changes from *idling* to *reading* and then returns to *idling* at step 2. At step 3 the DD receives the WRITE I/O request, triggering a switch to the *writing* mode. At step 4 the DD switches to the mode 0110 representing synchronous *reading*

and *writing*. After a while, the WRITE activity finishes and the DD changes to the *reading* mode as the READ routine has not returned yet (step 5). Eventually, the READ routine returns and the DD is *idling* again, waiting for new I/O requests (step 6).

The set of highlighted modes and transitions represent the *operational state space* of the respective DD. This set depends on (a) the I/O request types received by the DD and (b) the sequence of incoming / outgoing I/O requests. That is, the OSS varies with the workload used to exercise the DD.

Definition 4 (**Operational State Space**). *The* operational state space *(OSS) of a DD with respect to a workload is the set of modes visited (together with the traversed transitions) in the time interval spanning the workload execution.*

An important note has to be made at this point is that the OSS differs fundamentally from the *operational profile* of the DD, as defined by Musa [Musa, 1993]. According to his definition, the profile is "*a set of disjoint (only one can occur at a time) alternatives with the probability that each will occur*". Hence, the OSS only defines the "disjoined set" of alternative DD executions, but it does not associate probabilities to each of the modes or transitions. The OSS is enhanced into an operational profile (similar to Musa's definition) in Chapter 5, after a set of occurrence and temporal quantifiers are developed to capture the probabilities of each mode and transition.

Next, we discuss the value of the OSS as guidance for subsequent testing tools. A set of coverage metrics are developed to help quantify the progress of an ongoing test process.

4.2 Coverage Metrics for Testing Drivers

Software defect prediction research showed that software faults are not uniformly distributed throughout the code; they tend to cluster in certain areas [Fenton and Neil, 1999; Möller and Paulish, 1993]. Assuming that different inputs to the DD trigger different functionalities inside it, a logical implication is that testing an input can be associated with a certain likelihood of finding a fault.

According to the state-based approach presented in this thesis, each DD mode is associated with the execution of one or more disjoint pieces of code. Subsequently, a tester can use our driver state model to test each functionality separately (see Figure 3.7, the modes on the level 1 are associated with servicing single I/O requests) and in a combined manner (lower levels in the same figure).

It makes sense to test the modes located on levels below 1 ($l \geq 2$) as they might contain faults that surface only when a functionality is executed in conjunction with others, for instance faults triggered only when accessing shared resources. Therefore, a complete testing coverage of the OSS defined by our methodology is highly desirable because it represents a quantifiable measure of the test status. Moreover, no modes and no transitions outside the OSS need to be tested.

As long as a DD is in a certain mode, it can only switch to a limited number of neighbors located on the immediately upper and lower levels. A testing campaign aimed at transition coverage must exercise the entire set of transitions exiting any mode belonging to the OSS.

Ensuring traversal of all transitions belonging to the OSS can reveal errors that occur at entering or exiting a DD routine. For instance, mode 1000 in Figure 4.1 can be visited even if the transition 0000→1000 was never traversed (i.e., instead the path 0000→0100→1100→1000 was followed). Intuitively, this means that some of the transitions between modes might never be traversed even if all modes were visited, so *transition coverage* is more complete than *mode coverage* testing but is still not complete enough.

As DDs use memory to communicate and store variables while processing I/O dispatch routines (are *stateful* systems), the *sequence* of I/O requests that put the DD in a certain mode is relevant for the testing process, too. In the example above, one should be aware of the fact that the different paths between 0000 and 1000 may lead the system into two different states (i.e., having different memory contents). Our method cannot capture the whole driver state from this perspective (see the discussion in Section 3.3), but is useful from a testing viewpoint as it can capture the effect of input sequences like ordering of requests. Knowing the paths between two modes (from inspecting the OSS), a tester can develop test cases to traverse all of them in order to discover faults occurring as a result of certain input sequences. For instance, the sequence READ→CLOSE might work out fine in contrast to the sequence CLOSE→READ which might yield an error.

For complete test coverage of our OSS model a testing process should ensure that (a) *mode coverage*, (b) *transition coverage* and (c) *path coverage* are satisfied at the same time.

4.2.1 Mode Coverage

Our OSS model improves the granularity of the testing process: the WORKING mode of a DD as illustrated in Figure 3.5 is split into several, refined modes (see Figure 3.7, all modes located on levels $l \geq 1$). Using our method, one can precisely pinpoint which functionality the DD is executing at any

given time instant, with regard to the received I/O requests. This finer granularity contrasts with the generally impractical insight – "*the DD is currently working*" – unfortunately characterizing many black-box level test tools.

Moreover, the OSS model captures the concurrent execution of several I/O requests, so a testing campaign can identify faults in DD routines that only surface when the DD is executing them in conjunction with other procedures (the modes located on levels $l \geq 2$ in our OSS – see Figure 4.1).

If complete mode coverage is intended, a testing process can use test suites which force the DD-under-test into every OSS mode, thus implying that all the functionalities are executed at least once, separately *and* in conjunction with other ones.

Definition 5 (**Mode Coverage**). *A testing technique having 100% mode coverage (MC) ensures that every mode of the OSS is tested.*

The MC of a testing procedure can be quantified by relating the tested modes and the total number of modes forming the OSS, as a percentage:

$$MC = \frac{|\text{tested modes} \cap \text{OSS modes}|}{\#\text{ of OSS modes}} \cdot 100 \quad [\%] \tag{4.1}$$

4.2.2 Transition Coverage

Since the test space is not large (WDM defines 28 distinct IRP types, see Dekker and Newcomer [1999]; Oney [2003]; Orwick and Smith [2007], and not all modes are visited), covering all of it is feasible given that a set of test cases capable of putting the DD into each mode can be devised.

However, the MC metric only measures the test coverage of the IRP-related functionalities in all possible combinations, without considering how a DD leaves the current mode. Therefore, we need a more comprehensive coverage metric and, accordingly, we introduce the concept of *transition coverage*.

Definition 6 (**Transition Coverage**). *A testing technique having 100% transition coverage (TC) ensures that for each mode which belong to the OSS all outgoing transitions are tested.*

The progress in terms of TC of a given test procedure can be evaluated as a percentage using the following formula:

$$TC = \frac{|\text{tested transitions} \cap \text{OSS transitions}|}{\#\text{ of OSS transitions}} \cdot 100 \quad [\%] \tag{4.2}$$

Satisfying complete TC requires a larger number of test cases than MC but TC implies 100% MC, thus being a better measure of testing completeness.

4.2.3 Path Coverage

The concept of *path* is used in our approach to express a sequence of I/O requests that lead the DD under test from an *current* mode to a *destination* mode. Depending on the end modes and on the path length on hops, there might be several paths connecting the two modes. For instance, in Figure 4.1 there are two 2-hop paths from mode 1000 to 1110 (the number of paths is actually infinite if we consider cycles and multi-hop paths): **Path 1** – 1000→1100→1110 and **Path 2** – 1000→1010→1110.

Definition 7 (**Path Coverage**). *Path Coverage (PC) denotes traversing all paths between two different modes of the OSS (specified as* source *and* destination *of the paths), over any number of hops.*

We consider that the number of paths between any two modes in our OSS model is theoretically infinite. However, the PC can be used to compare the influence of following different paths between two modes, assuming that the parameters which differ can be captured. For instance, if a value at a memory location associated with the tested DD is different for the two paths, this might indicate a fault on one of the two paths. Of course, the semantics of the respective value has to be known to claim that the differing value is the result of an activated fault, a situation specific to white-box level testing.

4.3 Operational State Space Hypotheses

An important note here is that the test coverage metrics presented in the previous sections refer only to the operational state space and not to the total state space of a given DD. This differentiation implies that the metrics are related to the actual size of the OSS which varies depending on the workload used to obtain it. So, to be able to accurately measure the test progress (and, implicitly, the amount of pending testing) at a given time instant, a clearly defined OSS for the tested DD must be available.

In this section we introduce several work hypotheses that substantiate the differences between the OSS and the total state space of the DD. They are first intuitively presented and then evaluated via investigative experimentation using an actual Windows XP DD in combination with multiple workloads.

4.3.1 Hypothesis H1: OSS and Total State Space Size

We believe that the OSS forms only a very small part of the DD's total state space, irrespective of the workload used to obtain the OSS. This has some significant implications on testing coverage.

First, the OSS indicates the modes and transitions having high likelihoods to be reached in the field. This set of modes and transitions vary across several OSSs, but at least the subset represented by the intersection of several OSSs is very likely to be executed in the field.

Second, we believe that only few I/O requests can be serviced concurrently, due to DD and OS restrictions (i.e., access to shared resources). By exhaustively testing all combinations, the internal constraints on how requests can be concurrently executed might be disclosed. Based on the earlier assumption that not all DD modes are reachable, it implies that the total number of test cases used by in test campaign can be reduced dramatically, by filtering out the superfluous ones.

Third, assuming that a testing tool always starts applying test cases in the *idle* mode, the length of the sequence needed to bring the driver into the mode of interest for testing is very short as some I/O requests are mutually exclusive.

4.3.2 Hypothesis H2: Access to Lower-level Modes

As a direct implication of the previous assumption, we also hypothesize that the modes located on levels $l \geq 2$ in the OSS are have a smaller likelihood to be visited. This assumption is based of several facts, discussed below.

First, the execution of the I/O dispatch routines of a DD has to be as fast as possible in order to avoid any unnecessary delays occurring in the I/O path, notorious for its inherent slowness. This is recommended also as part of the DDK [Oney, 2003]. Therefore, the modes on levels $l \geq 2$ are visited only if two conditions are fulfilled at the same time: (i) the rate of the incoming I/O requests is higher than the completion rate and (ii) the execution of the incoming I/O requests is not excluded (or postponed) by the currently running I/O dispatch routines.

Secondly, inside the OS kernel rules of thumb recommend DD developers to keep the number of threads as low as possible in order to avoid race conditions, deadlock situations etc. [Dekker and Newcomer, 1999]. Moreover, the DD must be able to service as fast as possible any incoming I/O request, so the DD code must block as little as possible. Therefore, we consider that very few different I/O requests are actually allowed to be synchronously executed for any DD.

4.3.3 Hypothesis H3: Unequally Visited OSS Modes

Another hypothesis that deserves mentioning is that there is a significant difference between the visits to different OSS modes. This directly follows from H2 and has a deep implication for testing, which therefore has to treat the modes differently.

For instance, the OSS modes visited very frequently under a given workload have a high likelihood to be also visited very often in the field. Intuitively, the modes belonging to this category might be the modes associated with repetitive operations as READ or WRITE. Hence, the subsequent testing tool should primarily test those modes.

Similarly, the OSS modes visited infrequently should not be disregarded, as they are usually associated with infrequent but critical operations of the DD. For instance, the CREATE or CLOSE operations are executed only once, when the DD loads and unloads itself from the OS kernel, but they are central to the correct functionality of the DD.

4.3.4 Hypothesis H4: Testing Entails Accurate Profiling

Using the OSS by itself for testing purposes is not very useful. The reason is that the OSS represents only a subset of visited DD modes and traversed transitions out of the total state space of the considered DD.

By using the OSS, testing has access to an important source of information not available without it, namely the explicit bordering of the states which deserve focused attention from the rest of the DD states. Unfortunately, this insight is binary in nature (i.e., only *visited* or *not-visited*), thus a further prioritization of the test across the visited states is not possible. The use of the operational mode quantifiers introduced in Chapter 5 alleviates this issue.

4.4 An OSS Case Study – The Serial Driver

To validate the presented approach for building DD OSS, we have conducted experiments that monitor the flow of I/O requests sent to a targeted DD. The purpose is to investigate the hypotheses discussed above, in Section 4.3. Hence, we try to determine the shape and size of the OSS in contrast with the total state space of the DD.

For our experimentation we considered the serial DD provided as a part of the Windows XP Professional SP2. The executable image of the DD is

represented by the *serial.sys* file, version 5.1.2600.2180. The chosen DD is part of the default set of DD, thus being present on most machines running XP with Service Pack 2. It is used as both a function driver for legacy PnP and COM ports, or as a lower filter driver for PnP devices requiring 16550 UART interface.

As the selected DD successfully passed Microsoft's quality tests, it was digitally signed by Microsoft. This indicates that it was tested with the DD test tools available in the Windows Logo Kit (WLK) [Microsoft Corp., 2009b] and in the DDK (Driver Development Kit) [Oney, 2003], which includes reliability and stress tests.

For the experiments presented in this section we utilized two Pentium4(HT)@2.80Ghz machines with 1Gb of RAM each and a 56k external serial modem, Devolo Microlink 56k Fun II [Devolo, 2009]). To monitor the I/O request flow we used *IrpTracker* v2.1, a free tool from Open Systems Resources Inc. [Open Systems Resources, Inc., 2009]. This tool is capable of logging all the communication taking place at the I/O interface of the targeted DD. Both the incoming and outgoing I/O requests are logged.

The experimental setup is depicted in Figure 4.2. For each of the two experiments presented in this section, a different application was used as relevant workload for the serial DD. The workload produces I/O requests (via the I/O Manager) that triggers mode switches of the DD.

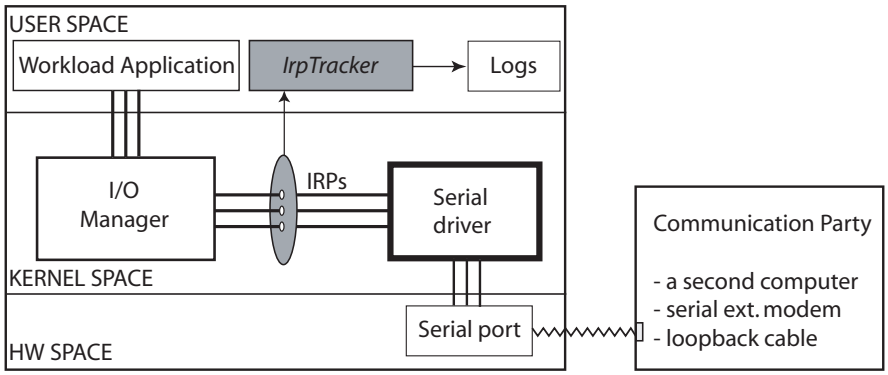

Figure 4.2: The experimental setup. The workload application exercises the serial DD (via I/O Manager) by communicating data through the serial port. I/O requests are captured and logged by *IrpTracker*.

We assume the DD is already installed in the OS kernel and properly initialized, so each experiment started with the DD in *idle* mode. After the IRP requests were captured and logged by the *IrpTracker* tool, we parsed and analyzed the log files sequentially and we built mode graphs similar to the one shown earlier in this chapter, in Figure 4.1.

4.4.1 Experiment 1 – Determining the OSS

For this experiment we used an external modem, connected to the serial port. As workload we used *ModemTest v1.3*, a diagnostic test from PassMark Software [Passmark Software, 2007]. The main purpose was to validate the viability of obtaining OSS graphs for combinations of DDs and workloads.

ModemTest sends data packets which are echoed back by the modem. Before sending any data, *ModemTest* first checks the serial port settings and then the modem itself. The received data is verified to ensure its completeness and correctness. We have chosen this diagnostic tool as it generates workloads which are representative for modems and thus for serial port usage. At the same time, we used the diagnostic tool as it generates a repeatable workload, a key requirement to ensure the repeatability among different experiment runs.

The observed behavior of the serial DD under this workload is represented by the mode graph shown in Figure 4.3. For readability only the transitions between visited modes were depicted. Details can be observed in Figure 4.4, which shows only the OSS (visited modes and traversed transitions).

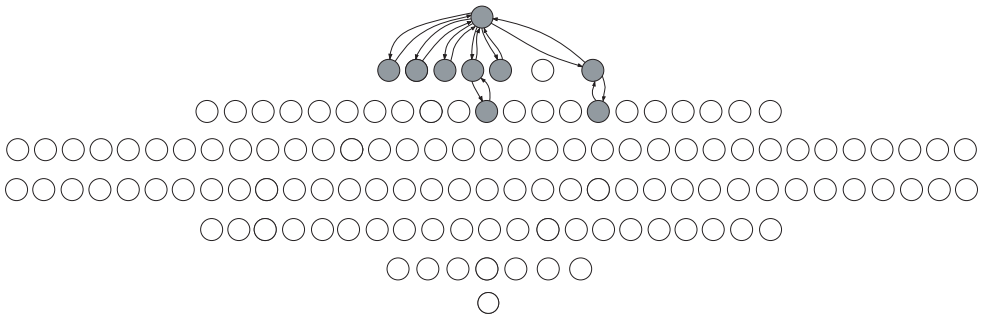

Figure 4.3: The obtained OSS vs. the total state space for the serial DD – *ModemTest* combination. The OSS is represented in gray. Note that the OSS represent only a small fraction of the total states space.

The experiment issued a total of 186 I/O requests (incoming and outgoing) using 7 distinct I/O requests (in order of appearance: CREATE, POWER, DEVICE_CONTROL, WRITE, READ, CLEANUP and CLOSE). With a total state space having 128 modes (2^7) and 896 transitions ($7 \cdot 2^7$), only 9 modes and 16 transitions were actually visited, forming the OSS for the considered workload. This means that only 7% of the modes and only 1.8% of transitions were traversed. Therefore, for properly testing the selected DD for the given workload, one has to focus only onto very few modes and transitions. This observation confirms our H1 hypothesis (see Section

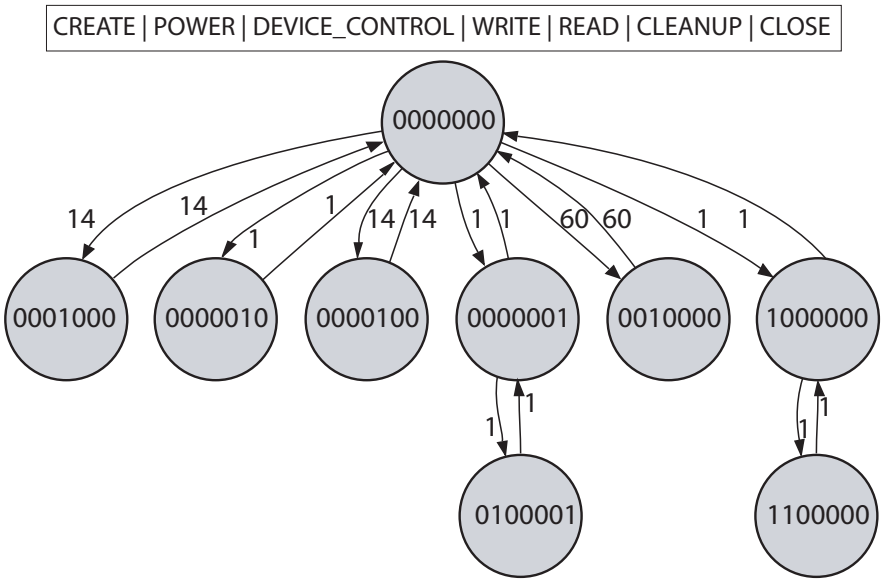

Figure 4.4: The OSS obtained for the *ModemTest* workload. Edge label indicate the number of traversals. The upper box shows the corresponding I/O request names.

4.3.1).

To verify if the obtained OSS is stable across multiple runs, we executed the same experiment five times, each time using different baud rates (4800, 14400, 19200, 57600 and 115200). The rest of COM port settings were the default ones (8/N/1/no flow-control). The observed flow of I/O requests was identical each time and produced the same graph of visited modes (Figures 4.3 and 4.4). Moreover, the number of issued I/O requests was also unchanged for all runs.

These facts indicate that the workload application is following an operational pattern, with each execution generating the same sequence of I/O requests. Additionally, the serial DD under test performed in a deterministic manner (responded to requests in the same manner every time). If a single I/O request would have been dropped or the DD would have finished executing the associated code in a different period of time (so that the DD would have started processing another incoming I/O request), the OSS would have had a completely different shape.

If the OS, DDs and the set of used applications are known (or at least a subset of them), the system tester can first build behavioral patterns that describe the manner the OS and the installed applications exercise a DD (the OSS), as sets of visited modes and traversed transitions. If needed,

counters can be associated to each mode and transition to observe which are frequently visited or, respectively, traversed (as depicted in Figure 4.4).

4.4.2 Experiment 2 - Aggregated Workload

In the following experiment we show the usefulness of the OSS for subsequent testing, thus tackling the H4 hypothesis (see Section 4.3.4). Under the assumption that the tester can identify the set of applications with impact on the targeted DD, we collected a set of applications which we used to generate a workload for the serial DD (see Table 4.1). The main goal of this experiment is to investigate the variability of the OSS across several workloads.

Table 4.1: The applications used as workloads for the serial DD.

Short	Application	HW at COM port	Description
A1	`BurnIn Test Pro v4.0`	Loopback serial cable	Reliability and stability test; RTS, CTS, DTR, DSR test, cycling all the baud rates
A2	`DirectX Diag. Tool v9.0c`	Computer (via serial cable)	DirectPlay test; text messages exchanged between machines
A3	`HyperTerminal v5.1.2600.0`	Computer (via serial cable)	Exchanged messages and 50k files
A4	`ModemTest v1.3`	External modem	See Section 4.4.1
A5	`Win XP modem diagnostics`	External modem	Windows XP queries the modem to check its capabilities
A6	`Win XP Device Manager`	External modem and COM port	Device Manager scans for hardware changes; the serial port and the modem are queried
A7	`Dial modem off`	External modem	Tried to dial a number when modem is off

Similar to the previous experiment, we built the OSS of the serial DD for the selected application set. Table 4.2 contains the results of the aggregated workload experiment, as well the results for each application.

107456 requests were issued in total, out of which 10 were distinct, thus modes are represented by 10 bits. The total state space graph (not shown here for space reasons) has 1024 (2^{10}) modes and 10240 ($10 \cdot 2^{10}$) transitions. Only 17 modes and 35 transitions were visited, which corresponds to 1.66% of modes and 0.34% of transitions.

Table 4.2: The aggregated results of the considered workloads.

Application	IRPs		Total State Space (from which (%) OSS)	
	Issued	Distinct	Modes	Transitions
A1	98398	7	128 (7.03)	896 (1.78)
A2	220	8	256 (4.68)	2048 (1.07)
A3	6704	7	128 (7.81)	896 (2.12)
A4	187	7	128 (7.03)	896 (1.78)
A5	1326	8	256 (3.90)	2048 (0.87)
A6	146	8	256 (4.68)	2048 (0.92)
A7	476	8	256 (4.29)	2048 (0.97)
Aggregated:	**107456**	**10**	**1024 (1.66)**	**10240 (0.34)**

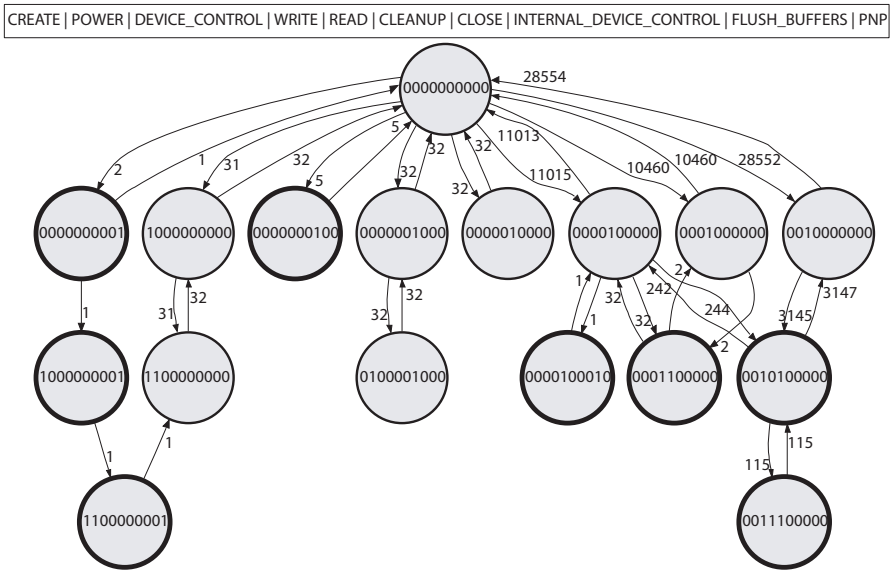

Figure 4.5: The OSS obtained using the aggregation of the applications described in Table 4.1. The thick-bordered modes were not visited by A4.

In Figure 4.5 we marked with thicker circles the DD modes visited under the aggregated set of workloads but not under the A4 workload, described in our previous experiment (see Figure 4.4).

Hence, this figure can be used to illustrate (obviously, at a lower scale) why testing fails to find all the defects present in the DD code. Assuming that DD test tools activate only the thick-bordered modes, defects present in the thin-bordered modes (assumedly activated in the field) have a high likelihood to surface, despite of the thoroughness of the used test methods.

Our experimental results show that even for a large set of varied applications only a very small percent of modes and transitions is visited, thus

confirming the hypotheses H1. To ensure a thorough activation of a large set of DD routines we have chosen a variate set of workloads, some of them serial port benchmarks, but still the set of visited modes was not proportionally larger than only one workload was used. This indicates that various workloads exercise the serial DD in a similar fashion (from the perspective of the visited set of modes – the OSS).

We observed also that the modes located on lower levels in our model are not visited; we have only two visited modes on level $l = 3$. This can be an indication of the low degree of I/O request interlacing, i.e., not many IRPs are permitted by the DD's design to execute concurrently, this confirming our hypothesis H2.

Another observation is the large difference between the number of times transitions and modes were visited (see Figure 4.5). That is, some modes were visited multiple times (i.e., the mode 0000100000 associated with WRITE operations), while others are only seldomly visited (i.e., the mode 0000000001 associated with PnP operations). This confirms our hypothesis H3.

4.5 Chapter Summary

By developing the concept of *operational state space (OSS)* this chapter established the necessary basis for introducing the operational profiling in the next chapters. The importance of the OSS for testing is discussed in terms of DD mode, transition and path coverage. Also, a set of four work hypotheses are presented as main aspects to be tackled by the ensuing experimental investigation.

Finally, this chapter experimentally validates the work hypotheses via two investigative experiments aimed at validating the viability of obtaining OSS for DD – workload combinations for actual Windows DDs. The presented experiments show that only a small subset of the driver modes are actually exercised by using several commercial applications (benchmarks) that generate workload for the DD monitoring sessions. This result is important, indicating the relevant areas to focus a testing method (the OSS). Therefore, using the OSS to guide testing represents a significant improvement over random test methods, given the tendency of faults to concentrate in specific areas of the code [Möller and Paulish, 1993].

Moreover, the OSS enables construction of *operational profiles* for the DD-under-test, associating with each mode and transition a probability of occurrence in the field as proposed in Weyuker [2003]. Additionally, the method of obtaining OSSs is non-intrusive, requiring no access to the source code of the OS and the DD under test.

Chapter 5

Operational Profiles

How to distinguish across the modes and transitions belonging to the operational state space? How can the operational profile of a device driver be obtained?

In the previous chapter we experimentally identified the OSS of the serial port DD provided with Windows XP SP2. The experimental results revealed that the reached modes and transitions and the selected DD are *consistent* across different runs. Moreover, the OSS of the studied DD has a *small footprint* (only 1.6% of the modes were visited and 0.3% of the transitions were traversed). Though small and stable, the applicability of the OSS for operational profiling purposes is limited as it divides the modes and transitions into only two subsets (gives only binary information i.e., *visited* and *non-visited*). Unfortunately, this is insufficient for a proper characterization of the DD's activity as the ability to distinguish among the visited modes and traversed transitions is missing.

As one of the main contributions of this thesis (**C3** – see Section 1.2.2), in this chapter we enhance the highlighted OSSs by introducing mode and transition quantifiers for an accurate characterization of the DD's runtime behavior. We start from the hypothesis that the higher detail level of the OSS quantification permits discovery and assessment of the existing execution hotspots inside DD's code (this approach constitutes the contribution **C4**). This assumption is justified for testing black-box DDs as the information about their runtime behavior is implicitly meager.

This chapter starts by introducing the concept of *operational profile* for DDs. Then, to support the construction of DD operational profiles, occurrence- and duration-based quantifiers are defined. A large-scale case

study for Windows DDs is also presented, alongside with the set of tools developed for capturing and analyzing the obtained operational profiles (contributions **C7** and **C8**).

5.1 Operational Profile of a Device Driver

In the previous chapter, the DD's OSS was defined as the set of visited modes and traversed transitions belonging to the DD's total state space under a workload exercising the respective DD. The concept of *operational profile* for DDs represents an augmentation of the OSS model with a set of metrics as an effort to enhance its value for testing.

To introduce the *operational profile* let us reuse the example of the DD that supports four distinct I/O requests (CREATE, READ, WRITE and CLOSE) described in the previous chapter, Figure 3.6. The leftmost bit of the tuple defining the DD mode is set as long as the respective DD performs the functionality associated with CREATE, the second leftmost bit is set while the DD performs READ and so forth. The DD can execute several activities of different types concurrently, in which case the binary string defining the mode contains several bits set.

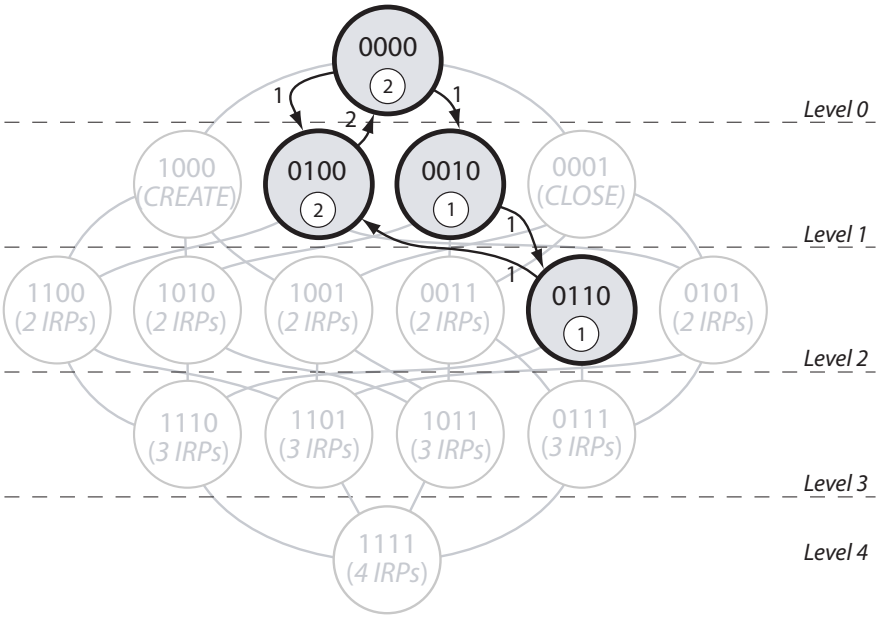

Figure 5.1: The *operational profile* of a DD supporting four IRPs. The operational profile adds probabilities to each of the OSS modes and transitions. The probabilities are computed using the mode and transition occurrence counters depicted as labels in this figure.

Figure 5.1 is structurally similar to the Figure 4.1 described in the previous chapter, the only difference lies in the presence of mode and transition labels in Figure 5.1. Transition labels indicate how many times the edges

were traversed, while mode labels (white circles inside the DD modes) represent the number of times the DD modes were visited. For instance, the label "1" of the transition 0000 → 0010 means that the respective transition was traversed once in the time frame described in the Figure 3.6, while the "2" on the mode 0100 means that the respective mode was visited twice.

The mode and transition labels in Figure 5.1 are obviously simple counters, but they play a key role in transforming a DD's OSS into a more useful concept for testing, namely the *operational profile*. The respective counters are later used to compute probabilities for each OSS mode and transition reached in the field, thus enabling the differentiation between the visited modes and transitions. This differentiation allows for prioritization of the test procedure, such that testing can be "tuned" on particular modes and transitions for various test scenarios.

Hence, we define the operational profile of a DD, while the next section develops the quantifiers for it, as probabilities.

Definition 8 (**Operational Profile**). *The* operational profile *(OP) of a DD with respect to a workload is the set of modes and transitions belonging to the DD's OSS together with their corresponding probabilities to be visited (or, respectively, traversed) in the time interval spanning the workload execution.*

Hence, the OP of a DD–workload combination is the OSS of the same pair, enriched with reachability probabilities for each of the modes and transitions belonging to the OSS.

5.2 Operational Profile Quantifiers

An accurate characterization of the operational behavior of a SW component is highly desirable for establishing effective testing methods. Our interest focuses on DDs, and these SW components are unfortunately known for their limited observability at runtime.

As test completeness is hard to achieve for DDs mainly due to their complexity, testers usually choose to primarily test the key functionalities [Mendonca and Neves, 2007]. Even when it is assisted by tools having a certain degree of automation, this selection remains a process based on tester's subjective experience in prioritization.

Hence, this section presents a set of quantifiers developed for differentiating among the visited modes and transitions of a DD's OSS (thesis contribution **C3** – see Section 1.2.2). Using these quantifiers, the relative frequencies of the visited modes and transitions can be observed and analyzed, revealing execution hotspots (thesis contribution **C4** – see Section 1.2.2).

The metrics introduced here are also useful as they provide accurate workload characterization from the DD's perspective. For instance, capturing the DD's field activity can be useful for workload assessments, usage/failure data collection or post-mortem debugging. Moreover, different workloads can be statistically compared to reveal the DD modes with higher probability of being reached in the field, thus allowing for a guided choice in the process of selecting the part of the DD to be targeted by the test tool.

From a testing perspective, the metrics associated with the DD modes indicate *how often* and *how long* each mode is visited. They represent key information about the operational mode of a DD for subsequent test campaigns, as they permit *inter-mode* priority rankings (*intra-mode* rankings are enabled by our DD code profiling methods presented in Chapter 7).

These rankings are tunable to the main purpose of the test scenario. For instance, if the goal is early discovery of the defects with high probability to occur in the field, then the test campaign should start by first covering the mostly visited DD mode, and continue in the decreasing order of the sojourn rate of the remaining modes until all of them are covered or the resources allocated for testing are depleted.

In this section we also introduce metrics for transitions among the modes belonging to an OSS. While we believe that the need for mode quantifiers for testing is intuitive (modes are abstract representations for the DD code), the arguments supporting the development of transition quantifiers are easier understood by using a simple example.

For instance, let us assume that a tester wants to test the mode 0011 in Figure 5.1. To "drive" the DD into that mode (onto the same transitions followed in the OSS!), one needs to design a test case which calls a WRITE shortly followed by a READ I/O request. A simple issuance of the two commands in this sequence might not be sufficient as the DD might finish the WRITE operation before the READ is called. If this happens, the test is applied to mode 0100 instead of the intended mode 0110.

We note that the test parameters alone (that is, of the issued I/O requests) cannot guarantee that the desired mode is reached. Consequently, a decisive test timing aspect is also involved. This means that the READ request must be called *before* the WRITE returns. Our transition quantifiers probabilistically capture the temporal dimension of the process, enabling the development of complex test cases (i.e., sequences of I/O calls instead of singletons). Hence, they permit computing the probability to reach the mode of interest depending on the current mode. Specifically, this probability is re-calculated in terms of the current mode after each hop in a sequence of I/O calls.

The developed metrics have a statistical meaning specifically in the con-

text of the workload for which they were assessed. Within this perspective, the OP can be used to detect deviations from the expected behavior of a DD by observing a population of runs of the selected workload and finding the OP which diverges from the rest of the runs in terms of one (or multiple) quantifiers. Chapter 6 illustrates the workload comparison procedure enabled by our approach.

For testing purposes, our metrics can be used to quantitatively compare the effectiveness of test cases (or test suites) on the DD-under-test. For instance, if the DD source code is not available, one can select the test cases (suites) having the highest coverage in terms of reached DD modes. Hence, while still reaching the same DD functionalities, the size of test case pool can be reduced to a least necessary minimum by removing the redundant test cases.

5.2.1 Occurrence-based Quantifiers

Two important characteristics of the runtime behavior of a DD are the mode and transition *occurrence weights*. They reflect the DD's likelihood to visit the mode (or transition) to which these quantifiers are bound. To express them, we first define as prerequisite notions the transition and mode *occurrence counts* for a given workload.

OP Counters: Transition Occurrence Counter (TOC) and Mode Occurrence Counter (MOC)

As a DD's OP is defined as its OSS annotated with mode and transition probabilities, the modes and transitions belonging to the OP are a subset of the total state space. In the following, we use the graph notation introduced in Definition 3; the OP is a connected subset of the total state space digraph, defined as "*the set of modes* $M = \{M_1^D, M_2^D, \ldots\}$ *as vertices and the set of transitions* $T = \{t_1, t_2, \ldots\}$ *as edges. Each transition from* T *maps to an ordered pair of vertices* (M_i^D, M_j^D)*, with* $M_i^D, M_j^D \in M$*,* $i \neq j$ *and the modes* M_i^D *and* M_j^D *within a Hamming distance of 1 from each other*".

Definition 9 (**Transition Occurrence Count**: $TOC_{t_{i,j}}$)**.** *The occurrence count for transition* $t_{i,j} \in T$*, originating in mode* M_i^D *and terminating in mode* M_j^D *(*$M_i^D, M_j^D \in M$ *and* $i \neq j$*) is the total number of recorded traversals from mode* M_i^D *to mode* M_j^D*.*

Definition 10 (**Mode Occurrence Count**: MOC_j)**.** *The occurrence count of mode* $M_j^D \in M$ *is the number of times mode* M_j^D *was visited during the workload execution.*

$$MOC_j = \sum_{i=1}^{N_{OP}} TOC_{t_{i,j}} \tag{5.1}$$

Note that both the occurrence counters expressed above are defined for the duration of the workload. Counter variables associated with each mode and transition can be used to store their values. The TOC and MOC counters are next utilized to develop subsequent quantifiers accurately specifying the operational behavior of DDs, namely the *mode occurrence weight* and the *transition occurrence weight*.

OP Weights: Transition Occurrence Weight (TOW) and mode Occurrence Weight (MOW)

Definition 11 (**Mode Occurrence Weight**: MOW_i)**.** *The occurrence weight of mode $M_i^D \in M$ represents a quantification of the driver's likelihood to visit the mode M_i^D relatively to all other sojourned modes of the OP (N_{OP}).*

$$MOW_i = \frac{MOC_i}{\sum_{i=1}^{N_{OP}} MOC_i} \tag{5.2}$$

This metric is similar to the metric used for development of OPs for building reliable SW components proposed by Musa [Musa, 2004]. In contrast, our quantifier is specific to profiling the runtime behavior of kernel-mode DDs and its significance is coupled with the specific workload for which it was computed. If the chosen workload accurately mimics the manner in which the DD is used in the field, the obtained mode quantifiers accurately express the field conditions.

Using this metric in profiling the runtime behavior of a DD helps building test priority lists. For instance, the modes with higher MOW value represent primary candidates for early testing, as higher values of this quantifier indicate the functionalities of the DD which are most frequently executed. For the *idle mode* (the mode where the DD is not executing any IRP-related activity) this quantifier indicates the percentage of mode sojourns that put the DD in an idle state, i.e., waiting for I/O requests.

Similar to MOW but referring instead to the transitions between modes, we define the *transition occurrence weight* for each traversed transition belonging to the DD's OP for a given workload.

Definition 12 (**Transition Occurrence Weight**: $TOW_{t_{i,j}}$)**.** *The occurrence weight of transition $t_{i,j} \in T$, originating in mode M_i^D and terminating in mode M_j^D ($M_i^D, M_j^D \in M$ and $i \neq j$) is the quantification of driver's likelihood to traverse the transition $t_{i,j}$ when leaving the mode M_i^D.*

$$TOW_{t_{i,j}} = \frac{TOC_{t_{i,j}}}{MOC_i} \tag{5.3}$$

Thus, the occurrence weight associated with the transition $t_{i,j}$ indicates the probability that this transition is actually followed when leaving the mode M_i^D. Note that the probability of following a certain transition depends on the current mode. This information is relevant for estimating which mode is to be visited next, given that there is a one-hop transition in the OP between the current mode and the one whose reachability is to be calculated.

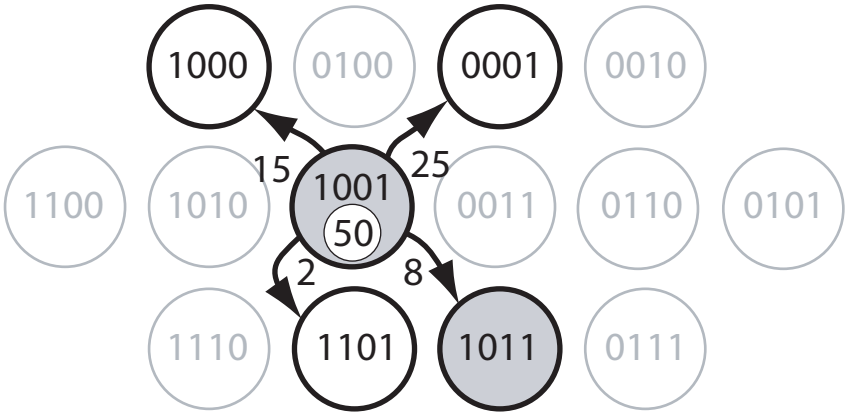

Figure 5.2: Calculating $TOW_{t_{1001,1011}}$

For instance, consider the situation depicted in Figure 5.2, where only the modes and the outgoing transitions of interest are shown, together with their TOC values as edge labels. The mode 1001 is current, the $MOC_{1001} = 50$ and $TOC_{t_{1001,1011}} = 8$. Therefore, $TOW_{t_{1001,1011}} = \frac{8}{50} = 0.16$. This indicates that the transition between 1001 and 1011 has been traversed 16% of the times the mode 1001 was left (i.e., 16% probability that the mode 1011 will be visited next when the mode 1001 is current).

5.2.2 A Duration-based Quantifier for Modes

To increase the accuracy of DD profiling, both spatial and temporal dimensions need to be considered. We regard the duration of a DD's activity not just as an artifact of the device's inherent slowness but as an important aspect

of the computation, as longer execution time reveals more defects (shown by many defect estimators in the field, e.g., the *Musa-Okumoto model* [Musa and Okumoto, 1984]).

Therefore, we introduce a quantifier accounting for the relative amount of time spent by the DD executing in each mode. As we consider the transitions between modes as instantaneous events, defining a corresponding temporal metric for edges is superfluous.

The overall time spent by the DD in each mode reveals information about the latencies of various I/O-related activities of a DD. If the DD spends a relatively large amount of time in a certain mode, that mode can be considered important for a subsequent testing campaign although the respective mode has a very low occurrence count (and, implicitly MOW).

For instance, the DDs managing "slow" devices as disks or tape drives spend large amounts of time in modes associated with READ or WRITE operations, irrespective of their sojourn rate. To capture this behavior we introduce a new OP quantifier, the *mode temporal weight*, to be used in conjunction with the mode occurrence weight for a multivariate characterization of DD modes.

Definition 13 (**Mode Temporal Weight**: MTW_i)**.** *The temporal weight of the mode $M_i^D \in M$ is the ratio between the amount of time spent by the driver in mode M_i^D and the total duration of the workload w.*

5.2.3 A Compound Quantifier for Modes

An accurate characterization of the runtime behavior of a DD needs to consider both the occurrence and duration-based metrics, on a stand-alone basis or in combination. In order to facilitate combinations of the two metrics, we propose a compound quantifier capturing these dimensions of the profiled DD's activity, namely the *mode compound weight*.

Definition 14 (**Mode Compound Weight**: MCW_i)**.** *The compound weight of a mode $M_i^D \in M$ is given by the expression (where $\lambda \in \mathbb{R}, 0 \leq \lambda \leq 1$):*

$$MCW_i(\lambda) = \lambda MOW_i + (1 - \lambda) MTW_i \tag{5.4}$$

By varying λ, MCW can be biased towards either occurrence or temporal dimension as needed for test requirements. For instance, to emphasize the temporal aspect λ should take values closer to 0, while the occurrence dimension is highlighted by values of λ that approach 1.

5.3 Experimental Evaluation

To validate the profiling approach and the associated metrics, we conducted a series of experiments that investigate the applicability of the OP quantifiers using a large number of Windows DDs.

Subsequently, the following sections introduce the used experimental setup and then present the set of DDs considered for the case study together with the applications used as workloads. Finally, detailed experimental data is presented and discussed.

5.3.1 Experimental Setup and OP Analysis Strategy

Our experimental procedure is a two-step process: first, the flow of I/O requests is captured and logged in a file (*online step*). In the second step the respective logs are analyzed and OPs are constructed (*offline step*). The two steps of the methodology for obtaining DD OPs are detailed below. For a graphic representation thereof, please refer to Figure 5.3.

Online step. To capture the I/O requests flow, we have built a lightweight "*filter driver*" interposed between the I/O Manager and a DD of our choice (Figure 5.3). This mechanism is widely used by many OSs for modifying the functionality of existing DDs or for debugging purposes.

Our filter driver acts as a wrapper for the monitored DD, logging only the *incoming* (from I/O Manager to DD) and *outgoing* (from DD to I/O Manager) IRPs. The I/O request traffic is then forwarded unmodified to the original recipient (the wrapped DD). As for logging each I/O request only a single call to a kernel function is needed, we expect the computation overhead of our filter driver to be marginal. The overheads introduced by our filter driver are assessed and discussed in Section 6.4.2.

Interposing our filter driver between the I/O Manager and a selected DD uses the standard DD installation mechanisms offered by Windows. Hence, it is non-intrusive and does not require detailed knowledge about the wrapped DD. The insertion and removal of the filter driver requires only disabling and re-enabling the target DD but no machine reboot. Moreover, due to its conformance to WDM, we used (*sans* modifications) the same filter driver to monitor all DDs whose runtime behavior was investigated in this thesis (see tables 5.1 and 5.3).

Offline step. To build the OPs we have designed the *OP-Builder* tool that processes the logs and outputs the DD's OP together with all runtime quantifiers. The figures 5.4 – 5.23 are obtained using directly the outputs of the *OP-Builder* tool.

For our experiments we utilized a Pentium4@2.66Ghz machine with

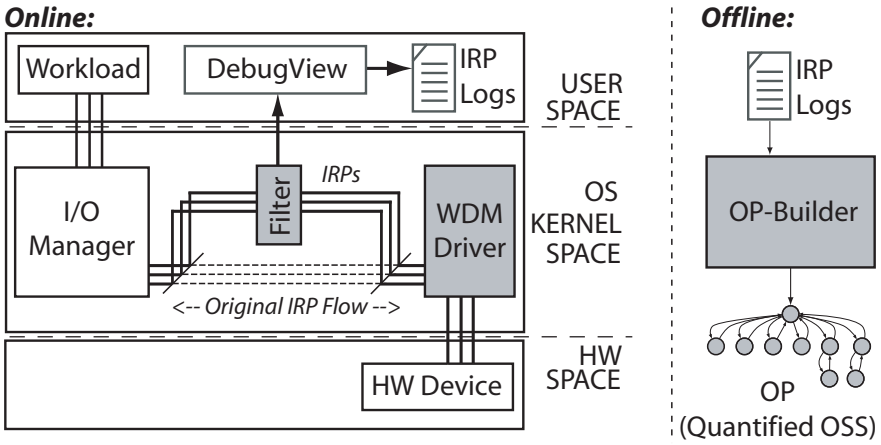

Figure 5.3: The experimental setup for obtaining the OP; the *online* (monitoring) and *offline* (analyzing) phases.

512Mb of DDRAM, equipped with Windows XP Professional SP2 (v5.01.2600). To build the filter driver we have used the tools provided by Microsoft as a part of the Windows Server 2003 DDK [Dekker and Newcomer, 1999; Oney, 2003]. For logging the kernel messages sent by the filter driver we have used Sysinternal's *DebugView* tool [Russinovich, 2008].

5.3.2 Studied Drivers and Workloads

As the case studies presented in this thesis are valid for WDM-compliant DDs, they are readily transferrable to Vista DDs, too. For its latest commercial OS (Vista) Microsoft introduced the Windows Driver Foundation (WDF) [Orwick and Smith, 2007]. WDF defines two main DD categories, the *user-mode drivers* (User-Mode Driver Framework - UMDF) and the *kernel-mode drivers* (Kernel-Mode Driver Framework - KMDF). As KMDF represents an extension of the earlier WDM, our case studies on WDM DDs hold also for Vista's KMDF-compliant DDs.

Currently, an important number of DDs are moving from WDM to KMDF as Vista's popularity is increasing. The error reporting facility of Windows Vista enabled Microsoft's researchers to estimate the unique devices attached to Vista OSs to be 390,000, while the DD population is increasing every day with 25 new and 100 revised DDs on average [Orgovan, 2008]. The vast majority of Windows DDs is represented by WDM/KMDF DDs [Microsoft, 2006], emphasizing the applicability of our profiling methodology presented in this thesis.

In this chapter we perform a systematic evaluation of the methodology

for obtaining the OP for five types of WDM-compliant DDs:

- a serial port DD (`serial.sys`) – for XP;
- a CDROM DD (`cdrom.sys`) – both for XP and Vista;
- an Ethernet card DD (`sisnic.sys`) – for XP;
- a floppy DD (`flpydisk.sys`) – both for XP and Vista and
- a parallel port DD (`parport.sys`) – for XP.

All DDs are provided (and digitally signed) by Microsoft, except the `sisnic.sys`, which is provided by the SiS Corporation. The DDs selected for the experimental evaluation presented in this chapter span all the different DD types described earlier in Section 3.1. Table 5.1 lists their main characteristics and features. Table 5.3 lists all the DD–workloads combinations used in the full spectrum of our experimental work.

To properly exercise the chosen DDs, we selected a set of benchmark applications generating comprehensive and deterministic workloads for the targeted DDs. For each experiment we analyzed the collected logs, constructed the OP graphs and calculated the OP quantifiers as previously defined in Section 5.2.

Beside commercial benchmarks testing the performance and reliability of the peripherals under various conditions, it is worth mentioning that we have additionally used the *Device Path Exerciser* (*DC2*) tool as a workload [Orwick and Smith, 2007, chap. 21, pp. 671–672].

DC2 is a robustness testing tool that evaluates if a DD submitted for certification with Windows is reliable enough for mass distribution. It sends the targeted DD a variety of valid and invalid (not supported, malformed etc.) I/O requests to reveal implementation vulnerabilities. *DC2* requests are sent in synchronous and asynchronous modes and in large amounts over short time intervals to disclose timing errors. In our experiments we exercised the parallel port and the floppy disk DDs with a comprehensive set of the *DC2* tests.

The results of experimenting with the chosen DDs are summarized in Table 5.2. In the full spectrum of our experiments all the other mentioned DDs also showed the effectiveness of our OP profiling method. The detailed results are presented in the next section.

Table 5.1: Considered DDs and their characteristics. The first column contains the short name used onwards to refer to the respective DDs, differentiating among DDs profiled under the Windows XP or Vista OSs.

Short	**Executable Image (Ver.)**	**Managed Device**	**Used as**	**Features**
`cdr_XP`	cdrom.sys (5.1.2600.2180)	DVD drive	Class driver, provides access to CD ROMs and DVD ROMs	PnP, power management and media change notification (autorun)
`cdr_Vista`	cdrom.sys (6.0.6000.16386)			
`flpy_XP`	flpydisk.sys (5.1.2600.2180)	Floppy drive	Block-device, legacy driver (monolithic)	Sits on top of the floppy disk controller in the driver stack mediating the communication with the user-level application which calls into the floppy disk controller
`flpy_Vista`	flpydisk.sys (6.0.6000.16386)			
`par_XP`	parport.sys (5.1.2600.2180)	Parallel port	Parallel port function driver and parallel port bus driver	PnP, power management, WMI, raw access to all parallel devices. Can share the access to all parallel ports on the system and detects all parallel enumerable devices connected to the port
`ser_XP`	serial.sys (5.1.2600.2180)	Serial port	Function driver for legacy PnP COM ports or as a lower-level filter driver for PnP devices requiring 16550 UART interface	PnP, power management, WMI. Controls interrupts and communication with device hardware (monolithic). Used in conjunction with `serenum.sys` (which acts as a device upper filter for `serial.sys`
`eth_XP`	sisnic.sys (1.16.00.05)	Ethernet card	*No information available (distributed as binary-only by the SiS Corp.)*	

5.3.3 Detailed Experimental Results

In this section we present the detailed results of the experiments listed in Table 5.2, in terms of the obtained OPs. All figures in this section (figures 5.4 – 5.23) are structurally similar, the information contained in the nodes of the OP graphs is the *mode name*, the *MCW value* and the *MTW value*. For instance, in Figure 5.4 the mode 000 has a MCW of 0.5008 and a MTW of 0.9951. Similarly, the transitions are marked with the TOW weight (see transition labels in the same figure).

Table 5.2: The workloads utilized to exercise the DDs and their experimental attributes, in terms of generated I/O traffic and operational profile size. The OPs captured for all the benchmark / DD / OS combinations listed in this table are presented in detail in Section 5.3.3.

Short	Benchmark	DD Short	IRPs		OSS		Description - Duration [min:sec]
			Issued	Types	Modes	Edges	
C1	BurnInTest-Audio	cdr_XP	1336	3 / 9	4	6	Audio CD test mode [07:40]
C2	BurnInTest-Data		71012	2 / 9	3	4	Data CD read / verify [01:03]
C3	BurnInTest-Audio	cdr_Vista	2170	5 / 9	6	10	Audio CD test mode [03:50]
C4	BurnInTest-Data		51034	5 / 9	6	10	Data CD read / verify [01:02]
F1	BurnInTest	flpy_XP	2946	5 / 6	6	10	Various read and write [05:03]
F2	Sandra Benchmark		19598	5 / 6	9	16	Perf. benchmarks [23:40]
F3	DC2		50396	4 / 6	5	8	MS DC2 tool [00:32]
F4	DevMgr-Disable		10	1 / 6	2	2	DevMgr – disable [00:0.01]
F5	DevMgr-Enable		400	3 / 6	4	6	DevMgr – enable drive [00:17]
F6	F1 – F5, sequentially		63396	5 / 6	6	10	Seq. run of F1–F5 [31:00]
F7	F1 ‖ F2 (run I)		12298	5 / 6	15	34	Conc. run of F1+F2 [18:40]
F8	F1 ‖ F2 (run II)		21884	5 / 6	15	34	Conc. run of F1+F2 [27:50]
F9	BurnInTest	flpy_Vista	3008	6 / 6	7	12	Various read / write [05:03]
F10	DevMgr-Disable		10	3 / 6	4	6	DevMgr – disable [00:.004]
F11	DevMgr-Enable		245	6 / 6	7	12	DevMgr – enable [00:0.03]
P1	DC2	par_XP	48530	6 / 6	7	12	MS DC2 tool [00:5.7]
S1	BurnInTest	ser_XP	11568	6 / 6	9	16	COM1 loop-back test [05:00]
E1	BurnInTest	eth_XP	2480	4 / 28	5	8	TCP/UDP full duplex [05:03]
E2	DevMgr-Disable		6	1 / 28	2	2	DevMgr – disable [00:01]
E3	DevMgr-Enable		114	6 / 28	7	12	DevMgr – enable [00:02]

Operational Profiles of the cdr_XP Driver

The workload C1 generated (Figure 5.4) 1336 IRPs belonging to 3 types (CREATE, CLOSE and DEVICE_CONTROL) out of a total of 9 types supported by the cdr_XP DD. The most accessed DD operation was DEVICE_CONTROL, the other two were only seldomly called.

Figure 5.5 illustrates the behavior of the cdr_XP DD under the workload generated by the BurnInTest application by testing a data CD media. 71012 IRPs were generated in total, belonging to 2 types (READ and DEVICE_CONTROL). READ was the most accessed DD operation under this workload, the DD spending here 82.35% of the total experiment time.

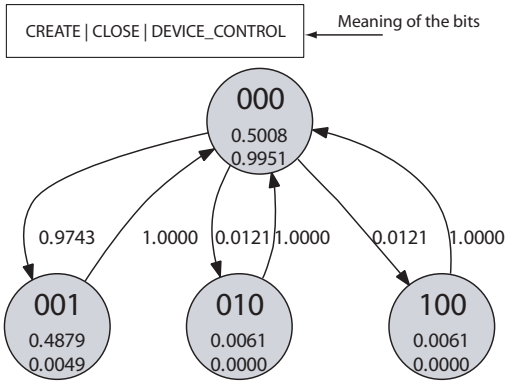

Figure 5.4: `cdr_XP` profile for the workload C1: BurnInTest - Audio

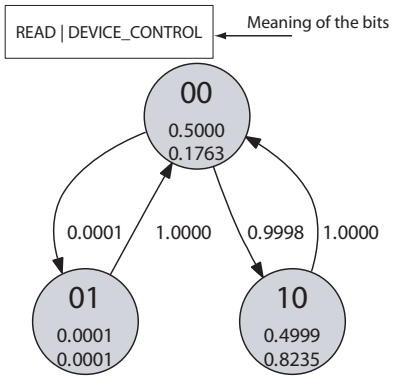

Figure 5.5: `cdr_XP` profile for the workload C2: BurnInTest - Data

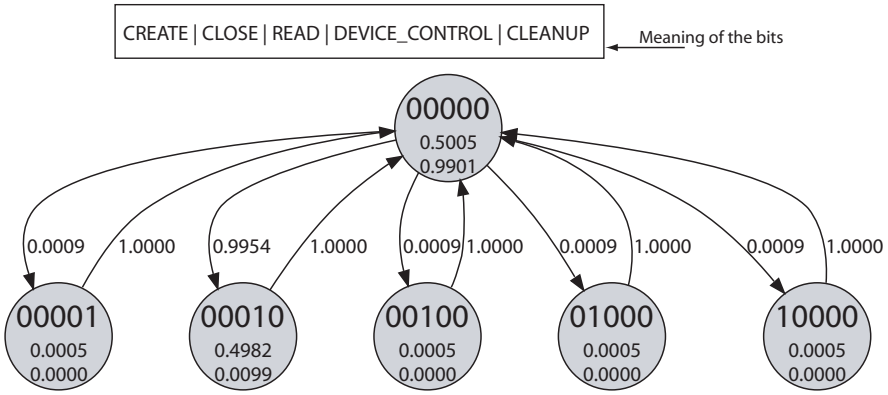

Figure 5.6: `cdr_Vista` profile for the workload C3: BurnInTest - Audio

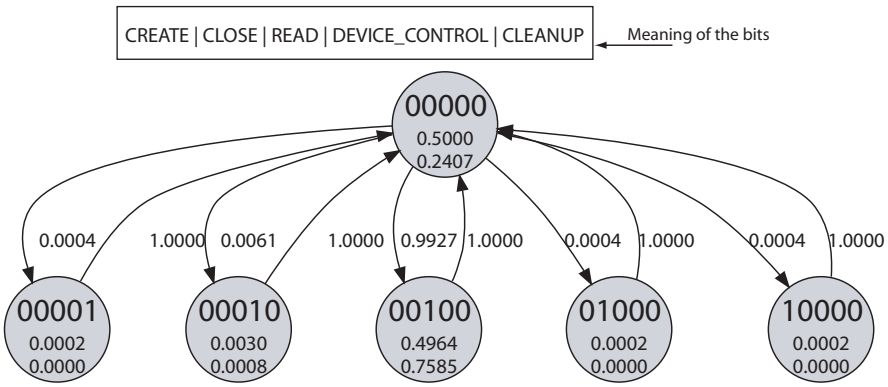

Figure 5.7: `cdr_Vista` profile for the workload C4: BurnInTest - Data

Operational Profiles of the `cdr_Vista` Driver

Under Vista, we re-used the same workloads that were used to exercise the DVD drive DD under Windows XP. However, we observed large differences in both shapes and quantifiers of the obtained OPs from figures 5.4 and 5.5 versus the figures 5.6 and 5.7.

These differences can be explained by the fact that different I/O Manager units build the I/O traffic for the responsible DD, as we are dealing with different OSs. For instance, the `cdr_Vista` DD received five types of IRPs from the Vista I/O Manager on behalf on both C3 and C4 workloads, while under Windows XP only three (and respectively, two) distinct types were issued (see Table 5.2). By comparing figures 5.4 and 5.6 in terms of issued IRPs, it becomes apparent that the `cdr_Vista` DD received two distinct IRPs (READ and CLEANUP) more than the `cdr_XP` DD. Figures 5.5 and 5.7 show the same trend, this time three distinct IRPs (CREATE, CLOSE and CLEANUP) were additionally issued for the `cdr_Vista` DD but not for the `cdr_XP` DD.

The *BurnIn Test* benchmark completed successfully for both XP and Vista. While the DDs there provide similar functionality, the difference in the obtained OPs indicate the different internal structures of the two versions of the considered DVD drive DDs.

Operational Profiles of the `eth_XP` Driver

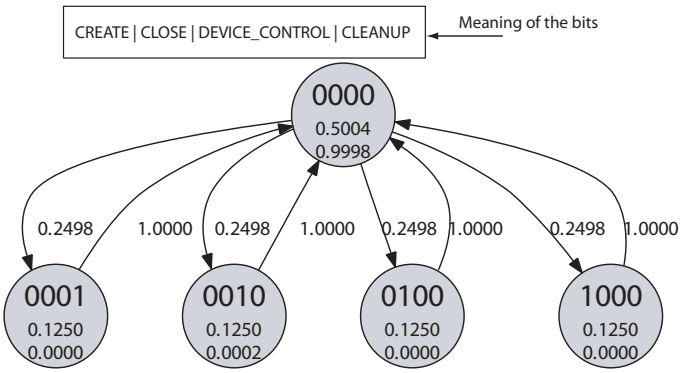

Figure 5.8: `eth_XP` profile for the workload E1: BurnInTest

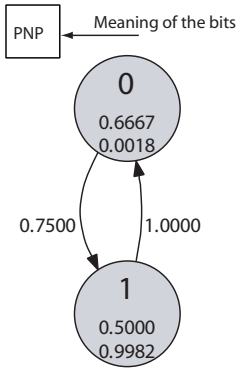

Figure 5.9: `eth_XP` profile for the workload E2: Device Manager - Disable driver

We exercised the ethernet DD (`eth_XP`) with three workloads. First we have used BurnInTest benchmark (Figure 5.8), followed by disabling the

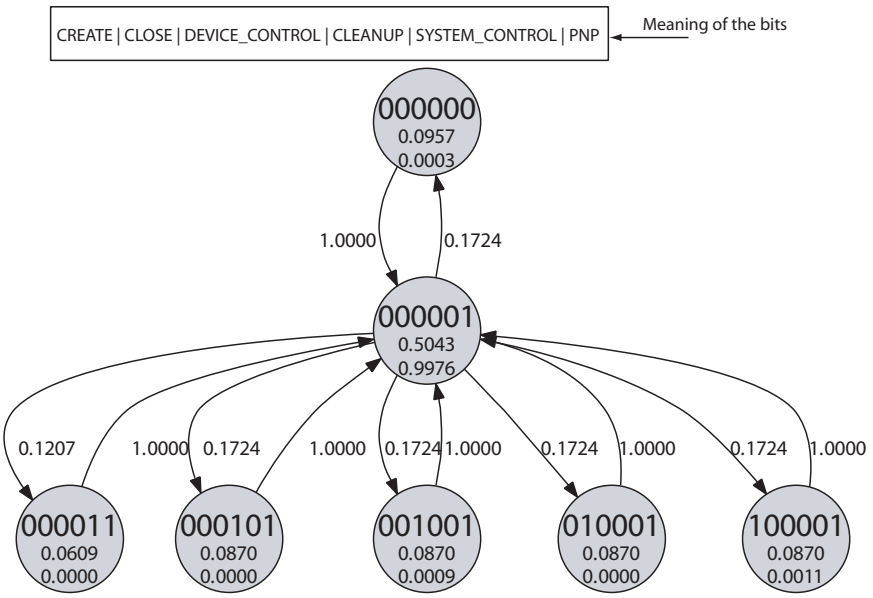

Figure 5.10: `eth_XP` profile for the workload E3: Device Manager - Enable DD

DD from the Windows Device Manager (Figure 5.9) and then re-enabling it (Figure 5.10). Therefore, figures 5.9 and 5.10 illustrate the activity performed by the DD at loading and unloading. This discloses the execution of the functionalities executed very seldomly (i.e., only at load and unload of the DD), but which are critical for the correct and dependable performance of the DD.

While the disable operation requires only the execution of the activity performed by a single IRP (CLOSE), the enable operation is more complicated, six IRPs are called, all of them in conjunction with the PnP activity. As both Enable and Disable operations require the PNP I/O operation to be executed, it indicates the high importance this IRP type has for the management of the DD in the OS.

Operational Profiles of the `flpy_XP` Driver

This section presents and discusses the OPs obtained for the `flpy_XP` DD by exercising it with the F1 – F8 workloads, as described in Table 5.2. The same `flpy_XP` DD along with the mentioned workloads are also the subjects of a different study (further presented in Section 6.2) aimed at revealing the ability to compare workload effects using our OP quantifiers.

The OPs induced by the workloads F1 and F2 onto the `flpy_XP` DD are

illustrated by the figures 5.11 and 5.12. The same two workloads were also selected to run concurrently, thus generating the F7 and F8 workloads.

BurnInTest accessed five IRP types without concurrent executions of the associated functionalities (Figure 5.11) in the time interval spanning the experiment. In contrast, Sandra benchmark called exactly the same IRP types, but in a manner that forced the DD into three additional modes located on the level 2 in the OP graph (Figure 5.12 - the modes 00011, 01010 and 10010). In all these three modes, the floppy disk DD accessed CREATE, CLOSE and DEVICE_CONTROL in conjunction with the WRITE operation. The mode 00010 (WRITE) is both the most accessed and the mode where the DD spent most of the time.

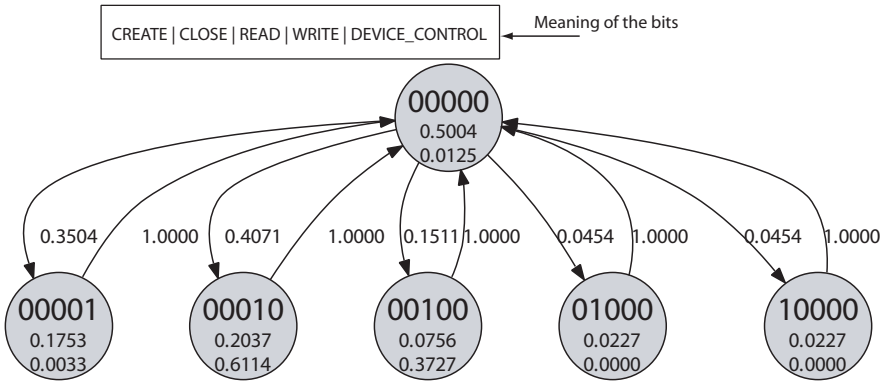

Figure 5.11: `flpy_XP` profile for the workload F1: BurnInTest

However, the behavior of the `flpy_XP` DD was counter-intuitive. *DC2* is a robustness test tool and given the fact that it generated a very large number of IRPs for the short interval of time it ran (averaging at more than 1500 IRPs per second), we had expected a large number of modes to be visited. Interestingly, the number of modes sojourned under *DC2* (Figure 5.13) was less than those for the workloads F1 and F2. Moreover, no modes associated with concurrent execution of DD functionality were visited. This might be an indication that the *DC2* waits until the DD finishes the current test then resets the device settings in order to start a new robustness test. This assumption is supported by the large number of times the mode responsible for DEVICE_CONTROL operations is accessed. Also, it indicates that *DC2* tests miss many operational modes of the DD.

Interestingly, for the floppy disk DD both the enable and disable operations use different IRPs than the Ethernet driver. Instead of issuing PNP I/O requests for disabling it, the I/O Manager calls CLOSE to disable the `flpy_XP` DD (Figure 5.14). Also, for enabling the DD (Figure 5.15), only

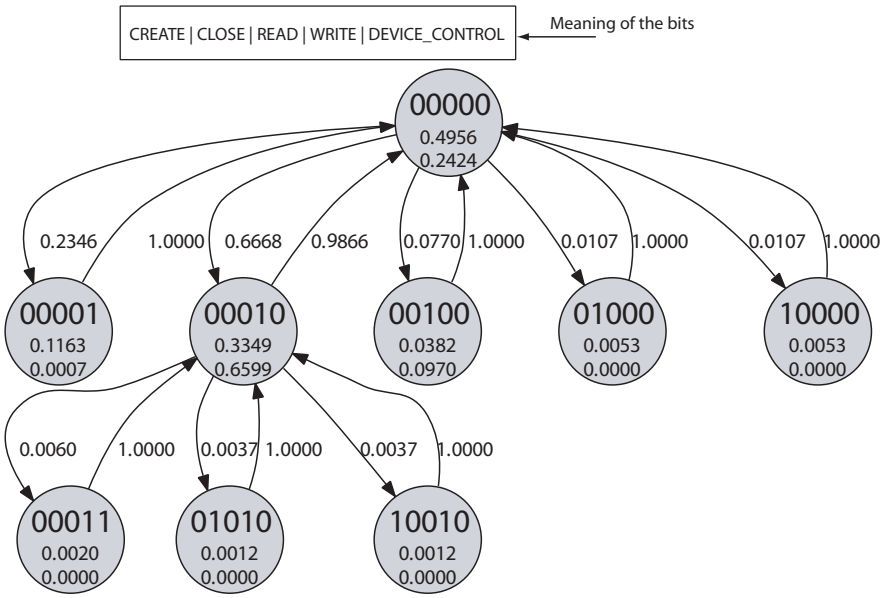

Figure 5.12: `flpy_XP` profile for the workload F2: Sandra benchmark

three IRP types are used (CREATE, CLOSE, DEVICE_CONTROL), an additional call to the PNP I/O request is not required. We believe that these major differences, in the manner in which DDs are loaded and unloaded from the OS originate in the functional and architectural variety among DDs following the WDM specifications.

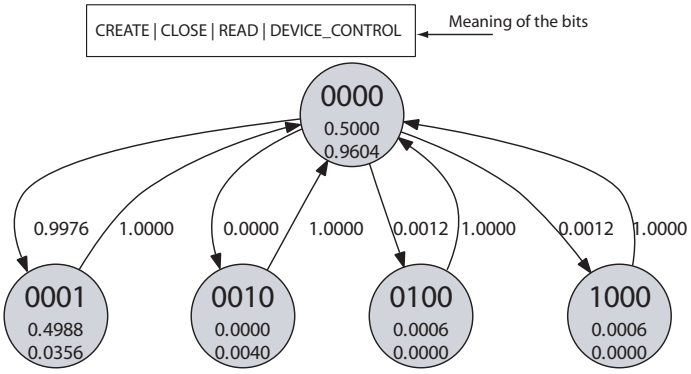

Figure 5.13: `flpy_XP` profile for the workload F3: DC2 (Device Path Exerciser)

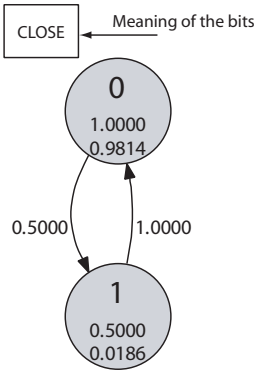

Figure 5.14: `flpy_XP` profile for the workload F4: Device Manager - Disable driver

Figure 5.16 is a graphic representation of the OP when the workloads

F1 to F5 were sequentially executed. The purpose is to study the runtime behavior of the `flpy_XP` DD over longer periods of time while the DD is subjected to multiple tasks. The DD executed mostly DEVICE_CONTROL operations in terms of number of sojourns to the respective mode, while WRITE was the most expensive operation in terms of time spent running the functionality associated with it.

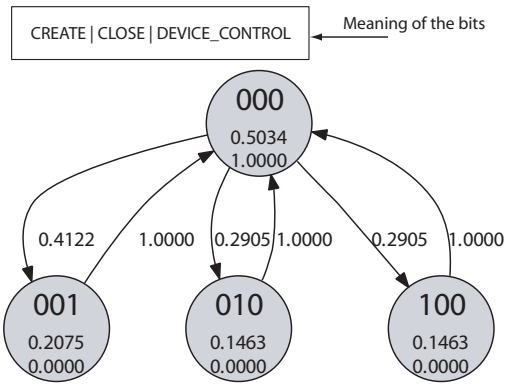

Figure 5.15: `flpy_XP` profile for the workload F5: Device Manager - Enable driver

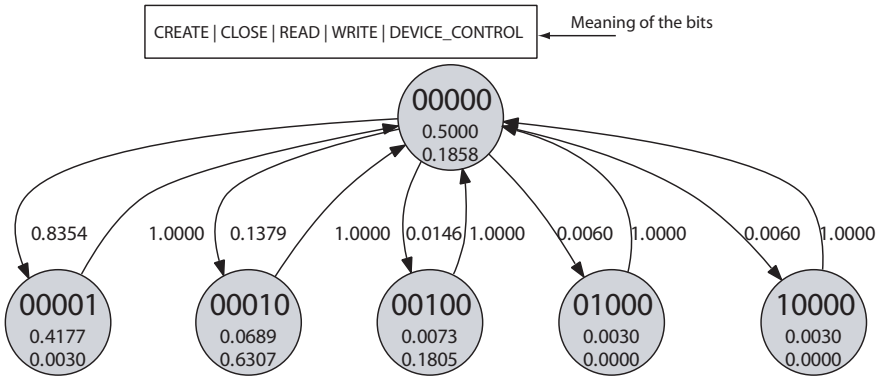

Figure 5.16: `flpy_XP` profile for the workload F6: Sequential execution of F1–F5 workloads

The concurrent executions of F1 and F2 revealed that exactly the same modes and transitions were visited irrespective of the order in which the workloads were started. The respective OPs are depicted in the figures 5.17 and 5.18. A detailed analysis and a quantitative comparison of these two workloads is presented in Section 6.2.

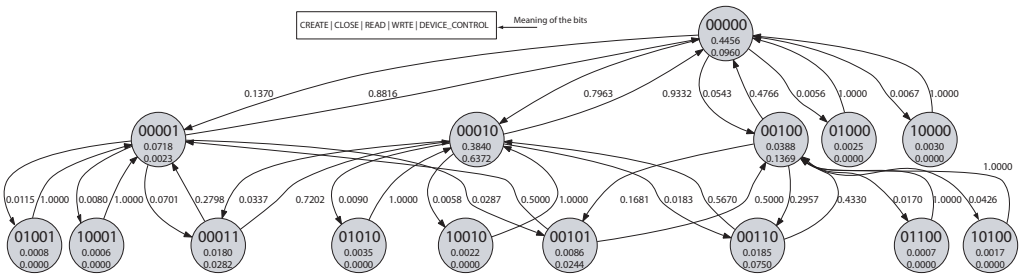

Figure 5.17: `flpy_XP` profile for the workload F7: Concurrent execution of F1 and F2 (run I)

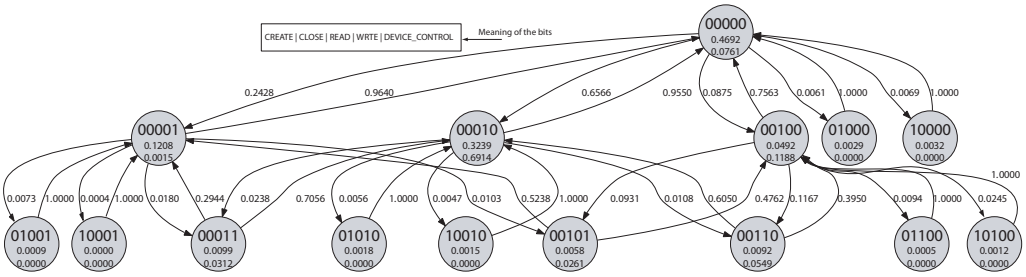

Figure 5.18: `flpy_XP` profile for the workload F8: Concurrent execution of F1 and F2 (run II)

Operational Profiles of the `flpy_Vista` Driver

For the DVD drive DD we conducted a detailed profiling of the operational behavior also for the floppy disk DD under Windows Vista and XP. The trend observed by comparing the two DVD drive DDs was confirmed by the comparison of the OPs of the floppy disk DDs installed under the two OSs. In general, under Vista more distinct I/O request types are issued than under Windows XP, while the benchmarks produced the same results.

For the `flpy_Vista` DD, on behalf of the BurnInTest benchmark, the Vista's I/O Manager issued additionally to the XP's I/O Manager the CLEANUP IRP (see figures 5.11 and 5.19). The structure of the OP graphs is similar, only the modes located on the first level were visited for both floppy disk DDs.

By comparing the OPs obtained for Disable operations (workloads F4 and F10, figures 5.14 and 5.20), we observed that two more distinct IRP types were issued for `flpy_Vista` than for `flpy_XP`, hinting at a more complex mechanism for unloading DDs in Vista than in Windows XP, even though exactly the same number of IRPs were issued under both OSs (namely 10,

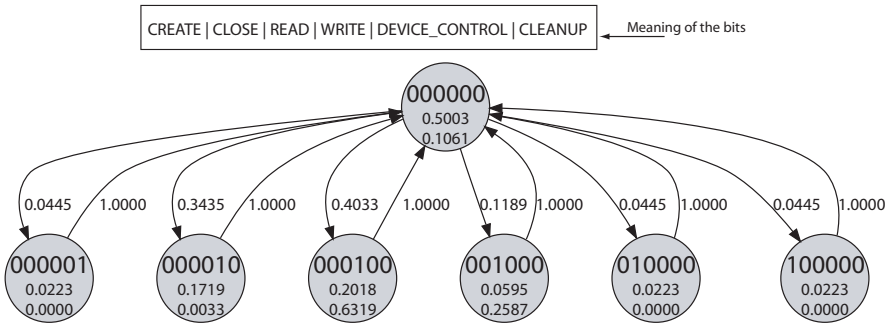

Figure 5.19: `flpy_Vista` profile for the workload F9: BurnInTest

see Table 5.2).

In contrast to the DD unload mechanism, by comparing the install routines of the two OSs (figures 5.15 and 5.21), we observed that Vista is much faster than XP (0.03 seconds for Vista versus 17 seconds for XP). The number of issued IRPs under XP was reduced in Vista to almost half, even though the distinct types they belong to contains three more in Vista (QUERY_INFORMATION, CLEANUP and PNP).

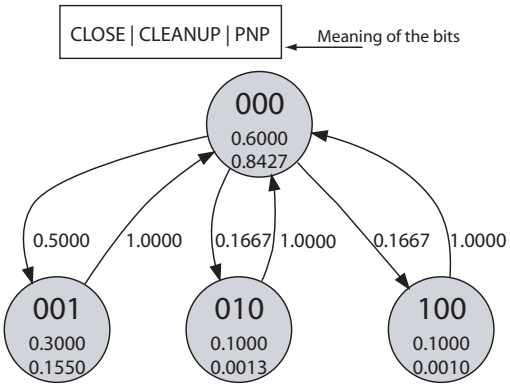

Figure 5.20: `flpy_Vista` profile for the workload F10: Device Manager - Disable driver

Operational Profiles of the `par_XP` Driver

Figure 5.22 illustrates the behavior of the `par_XP` using the *DC2* robustness tool as workload. Similar to the OP captured in Figure 5.13 for the `flpy_XP` DD, *DC2* only accessed modes located on the first level in the case of the `par_XP`, too. For the parallel port, *DC2* additionally called the QUERY_INFORMATION, CLEANUP and INTER-

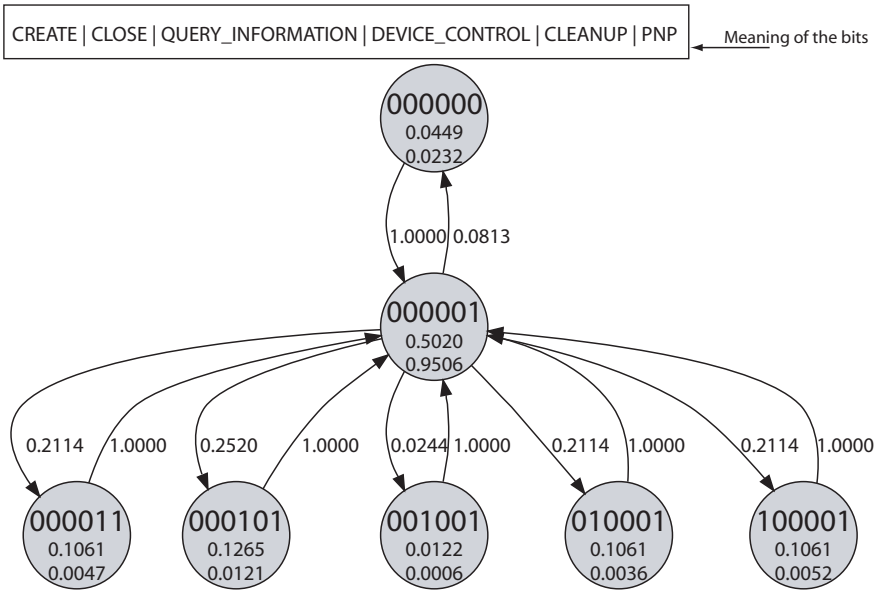

Figure 5.21: `flpy_Vista` profile for the workload F11: Device Manager - Enable driver

NAL_DEVICE_CONTROL, but it did not call the READ IRP. Overall, the functionality associated with the DEVICE_CONTROL operation was mostly visited, indicating that the setting and getting of device capabilities represents significant activity from the perspective of the *DC2* robustness test tool.

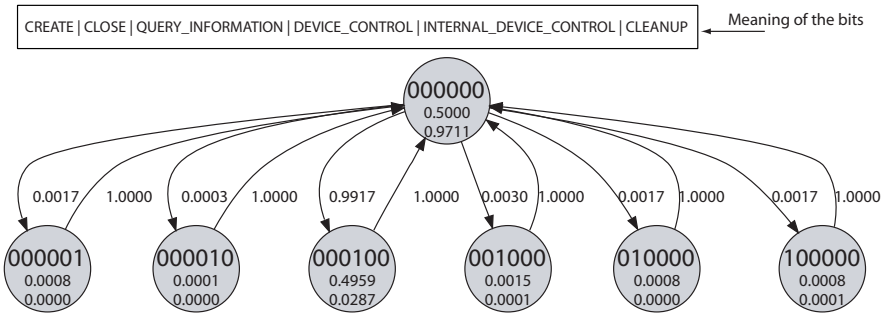

Figure 5.22: `par_XP` profile for the workload P1: DC2 (Device Path Exerciser)

Operational Profiles of the `ser_XP` Driver

Figure 5.23 graphically represents the OP recorded for the `ser_XP` DD under the BurnInTest workload. Only two modes (0100001 and 1000001) were

visited on the second level, most of the mode sojourns being made to the modes associated with READ (0010000) and WRITE (0001000) operations. The most of execution time was also spent in the WRITE mode.

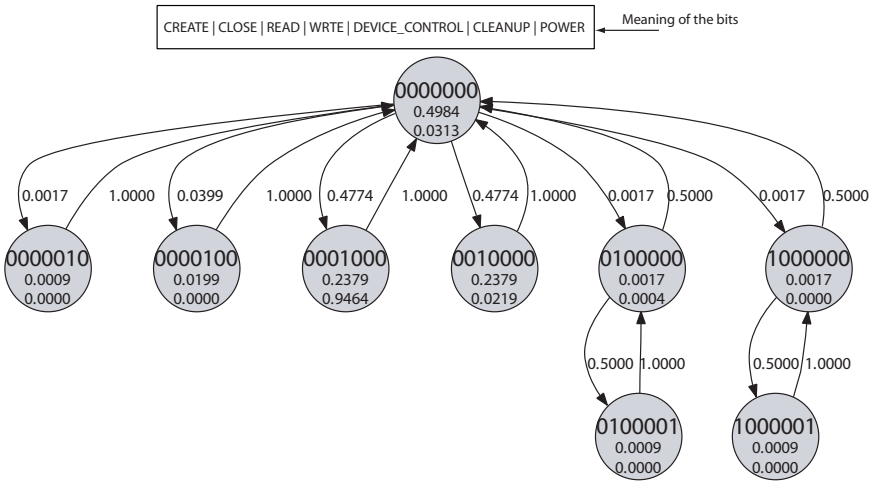

Figure 5.23: `ser_XP` profile for the workload S1: BurnInTest

5.3.4 Other Profiled Drivers

Due to space limitations, in the previous section we only presented in detail a subset of our experimental results. The complete set of DD for which OPs were built is listed in Table 5.3, while the obtained OPs are available online [Sârbu, 2009]. The list includes 50 DDs running on different versions of XP and Vista OSs, exercised using more than 260 workloads on more than 15 different machines. We also profiled several versions of the same DDs (for instance, see *cdrom.sys*, *flpydisk.sys* or *serial.sys* in Table 5.3).

Table 5.3: Other DDs profiled in our experimental evaluation. The obtained OPs are available online at [Sârbu, 2009].

#	DD Image	Version	Device	OS	# Workloads
1	rfcomm.sys	5.1.2600.5512	Bluetooth	XPSP3	3
2	cdrom.sys	6.0.6000.16386	CDROM	Vista	13
3	cdrom.sys	5.1.2600.2180	CDROM	XPSP2	15
4	cdrom.sys	5.1.2535.0	CDROM	XPSP2	20
5	cdrom.sys	5.1.2600.5512	CDROM	XPSP3	18
6	sisnic.sys	2.0.1039.1210	Ethernet Adapter	Vista	10

continued on next page →

← continued from previous page

#	DD Image	Version	Device	OS	# Workloads
7	b57xp32.sys	8.48.0.0	Ethernet Adapter	XP Home	3
8	pcntpci5.sys	4.38.00	Ethernet Adapter	XPSP3	8
9	pcntpci5.sys	4.38.00	Ethernet Adapter	XPSP2	2
10	sisnic.sys	1.16.00.05	Ethernet Adapter	XPSP3	8
11	flpydisk.sys	5.1.2600.0	Floppy disk	XPSP2	3
12	flpydisk.sys	5.1.2600.2180	Floppy disk	XPSP2	15
13	flpydisk.sys	6.0.6000.16386	Floppy disk	Vista	30
14	flpydisk.sys	5.1.2600.5512	Floppy disk	XPSP3	9
15	disk.sys	6.0.6001.18000	Hard Disk Drive	VistaSP1	3
16	disk.sys	5.1.2600.5512	Hard Disk Drive	XPSP3	2
17	parport.sys	6.0.6000.16386	Parallel Port	VistaSP1	3
18	parport.sys	5.1.2600.2180	Parallel Port	XPSP3	1
19	pci.sys	5.1.2600.5512	PCI Bus	XPSP3	3
20	pcmcia.sys	5.1.2600.5512	PCMCIA Adapter	XP Home	2
21	dot4prt.sys	5.1.2600.0	Printer	XPSP2	3
22	Dot4Prt.sys	5.1.2600.0	Printer	XPSP3	3
23	usbprint.sys	5.1.2600.2180	Printer	XPSP2	3
24	serial.sys	6.0.6000.16386	Serial Port	VistaSP1	3
25	serial.sys	5.1.2600.2180	Serial Port	XPSP3	1
26	ac97intc.sys	5.1.2535.0	Sound	XPSP2	3
27	adihdaud.sys	5.10.01.5410	Sound	XPSP2	3
28	ALCXWDM.SYS	5.10.5900	Sound	XPSP2	3
29	smwdm.sys	5.12.01.3533	Sound	XPSP2	1
30	drmk.sys	1.0.4825.0	Sound	XPSP2	3
31	tcpip.sys	5.1.2600.5512	TCPIP Services	XPSP3	2
32	AF15BDA.sys	8.6.24.1	TV Tuner	XPSP3	3
33	usbhub.sys	6.0.6000.16386	USB Controller	Vista	6
34	rndismpx.sys	5.1.2600.2781	USB Mobile Phone	XPSP2	3
35	hidusb.sys	5.1.2600.5512	USB Mouse	XP Home	1
36	usbhub.sys	5.1.2600.2180	USB Mouse	XPSP2	3
37	usbscan.sys	5.1.2600.1106	USB Scanner	XPSP1	3
38	disk.sys	5.1.2535.0	USB Stick	XPSP2	8
39	disk.sys	5.1.2600.5597	USB Stick	XPSP3	3
40	disk.sys	6.0.6001.18000	USB Stick	VistaSP1	3
41	usbstor.sys	5.1.2600.5512	USB Stick	XP Home	3
42	Volsnap.sys	5.1.2600.5512	USB Stick	XPSP3	3
43	ks.sys	5.3.2600.2180	USB Video	XPSP2	3
44	fwlanusb.sys	2.0.6.1647	USB WLAN	XPSP3	3
45	EU3USB.sys	2.1.1.0	USB WLAN	XPSP1	3
46	VboxVideo.sys	2.0.4.0	Video Adapter	XPSP2	3
47	BCMWL5.SYS	3.100.46.0	WLAN	XPSP2	3
48	bcmwl5.sys	5.10.38.26	WLAN	XP Home	3
49	NETw5v32.sys	12.1.0.14	WLAN	VistaSP1	3

continued on next page →

← *continued from previous page*

#	DD Image	Version	Device	OS	# Workloads
50	ar5211.sys	4.1.2.156	WLAN	XPSP3	3
DD–workload combinations additionally profiled:					**262**

5.4 Chapter Summary

This chapter introduced the concept of *operational profile* of a DD as the DD's OSS enriched with probabilities to be visited for each of the modes and transitions belonging to the OSS. To express these probabilities in a meaningful manner for testing, a set of occurrence- and duration-based quantifiers were developed for both modes and transitions. Moreover, a compound quantifier – the MCW – was introduced to ease tuning the subsequent test tools towards either occurrence or temporal probability of each DD mode.

Onwards, to validate the viability of the methodology for obtaining OPs, a set of seven XP and Vista DDs were exercised with several workloads. At the same time, their activity was captured using a specially-designed filter DD interposed between the I/O Manager and the targeted DD. The activity logs were parsed offline and the OPs of the selected DD–workload pairs were constructed. Finally, all the DDs used in the full spectrum of our experimental evaluation are listed.

Chapter 6

Operational Profiles' Usefulness

How do operational profiles help for improved testing of device drivers? What are the experimental issues that need be considered when building operational profiles?

The operational profiles are useful – as defined and experimentally obtained in the previous chapter – for multiple testing-related purposes. Onwards, we present a test prioritization methodology that uses the available operational profiles to guide the testing progress by producing a ranking of the DD modes in terms of occurrence and temporal weights thereof.

Next, a method for accurate workload cross-comparisons using the operational profiles is presented, constituting the contribution **C6** of this thesis (see Section 1.2.2). By highlighting the differences between the way the DD is exercised in the field against the test runs executed before the release of the DD, such a methodology enables a quantifiable assessment of the test accuracy and adequacy. The test space reduction of the OP against the OSS and the total state space of the DD are evaluated and discussed in this chapter.

The last part of this chapter presents several experimental aspects. Among these, notable are the validity of the experimental procedure, the operational overhead introduced by our DD monitoring mechanism and the effort required to profile OS DDs.

6.1 Test Prioritization via OP

Figure 6.1 depicts the Windows XP default floppy disk DD's OP, as exercised by the F2 workload. For detailed descriptions of the applications used as workloads – including F2 – see Table 5.2 in Section 5.3.2. Nodes are labeled with the mode name, MOW and MTW values (the latter in square brackets), while transition labels represent the TOW values of the respective directed edges. For simplicity, the occurrence counters of modes (MOC) and transitions (TOC) are not shown in the figure.

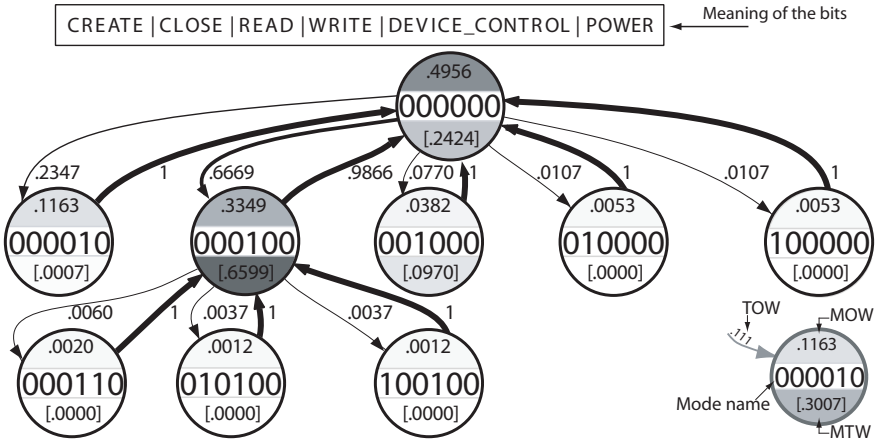

Figure 6.1: The OP for F2 and three prioritization cases; the figure is content-wise similar to Figure 5.12, with the difference that here we visually emphasized the execution hotspots by using darker shades and thicker lines.

In Figure 6.1 the darker gray hemispheres represent higher MOW or MTW values, revealing the frequently executed DD functionalities and the ones with longer execution times, respectively. The mode 000000 was predominantly visited under the workload, indicating that the DD was idle almost 25% of the time. Most of the time (66%) was spent by the DD executing the functionality of the mode 000100. This result is intuitive, as this mode is associated with the slower WRITE operation of the floppy disk drive.

Table 6.1: Priority ranking example for the modes in Figure 6.1

λ	**Priority Rank**							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
$\lambda = 0.25$	000100	001000	000010	010000	100000	000110	010100	100100
$\lambda = 0.50$								
$\lambda = 0.75$		000010	001000					

Depending on the testing purpose, the balance factor λ (see Eq. 5.4, MCW) can be tuned to guide test prioritization. Varying λ from 0 to 1 is equivalent to move emphasis from MTW to MOW. Table 6.1 represents the test priority ranking based on MCW of the modes when λ is 0.25, 0.5 and 0.75 (rank 1 has the highest priority). For instance, when $\lambda = 0.5$ (equal balance between MOW and MTW weights is desired), the mode 000100 is identified as a primary candidate for testing, followed by 001000 and 000010. In contrast, a $\lambda = 0.75$ (when MOW dominates MTW) keeps the same mode on the first priority rank but swaps the order of the second and the third modes. All other modes keep their ranking irrespective if λ takes the values of 0.25, 0.50 or 0.75.

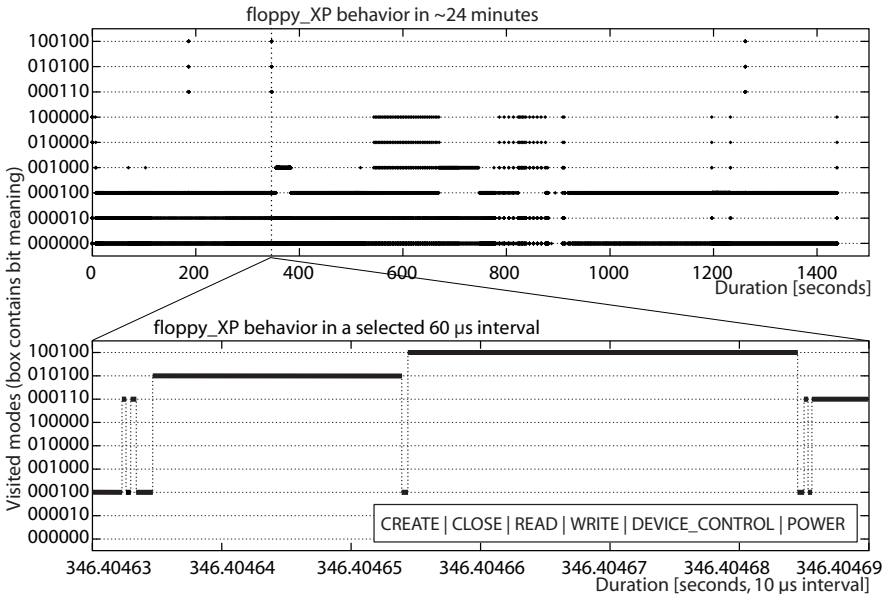

Figure 6.2: Temporal evolution for F2. Note that the DD is executing only in a single mode at any instant; this is more visible in the lower part of the figure, where the sojourn pattern of a very short interval of time is presented, with the transitions among modes depicted as vertical dotted lines.

Figure 6.2 illustrates the temporal evolution of F2 with different time windows. The upper part of the figure depicts the DD's evolution over the entire experiment duration (23 minutes and 40 seconds), where dots indicate mode sojourns. This representation reveals distinct execution patterns. For instance, the modes 000110, 010100, 100100 are visited only seldomly and also repeat (three times) the same sojourn pattern. For a better insight, the lower part of the figure shows the mode sojourn pattern in an arbitrarily selected 60μs time frame. This representation exposes the functionalities in

execution at any instant enabling testers to reproduce the IRP sequence that brings the DD in a mode of interest (i.e., to create effective test cases).

6.2 Workload Comparison via OP

Continuous post-release service provision is problematic mainly due to the limited capacity of the in-house testing to accurately reproduce the entire spectrum of possible field workloads. To estimate the difference between the behavior seen during in-house testing and field behavior, a comparison of the OPs can be performed. Ideally, a minimal difference ensures post-release test adequacy [Weyuker, 1998], though measuring it is not trivial without detailed knowledge about the field workload and a set of metrics to quantify the deviation. Assuming that several OPs of the DD are available, our runtime behavior quantifiers can be used to outline their relative deviation.

6.2.1 One-to-one Comparison

We present here a comparative study of two workloads for the floppy disk DD, workloads F7 and F8 (see Table 5.2). For F7, we first started the F2 workload, followed after a few minutes by F1. For F8, we swapped the start order. Both F1 and F2 are performance benchmarks that create, read, write and delete files on the floppy disk. Both F1 and F2 completed successfully when run concurrently despite the fact that F7 is 9 minutes and 9586 IRPs shorter than F8.

We observe that the structures of the OPs for F7 and F8 are identical (exactly the same modes and transitions were visited – see figures 5.17 and 5.18). We note that considering only the structure of the OP graphs when comparing workloads is not effective for runtime behavior profiling as the level of provided detail is limited. For instance, it is impossible to differentiate between a workload executing multiple READ operations and a workload accessing this operation only once, as the *reading* mode appears in both of the two obtained OPs. This fact recommends using the OP quantifiers for a more accurate comparison.

Figure 6.3 shows a side-by-side representation of the MCW ($\lambda = 0.5$) values for each mode, as generated by the F7 and F8 workloads. Similarly, Figure 6.4 depicts the TOW values side-by-side for each traversed transition. The values above the bars represent the absolute difference between the MCW values of each mode, and between TOW values for each transition. In both figures the small difference between the two OPs is apparent, despite of an almost double amount of issued IRPs for F8 relative to F7. The largest

deviations among MCW values are for modes 000010 (0.024) and 001100 (0.015), an indication that F7 executed more READ + WRITE operations and less DEVICE_CONTROL in contrast with F8.

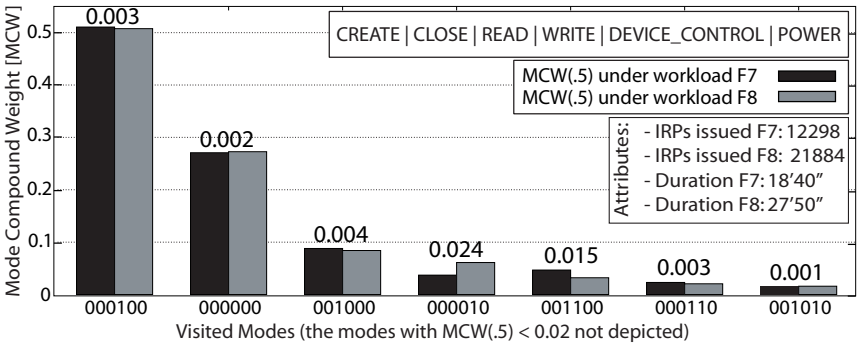

Figure 6.3: MCW comparison for F7 and F8

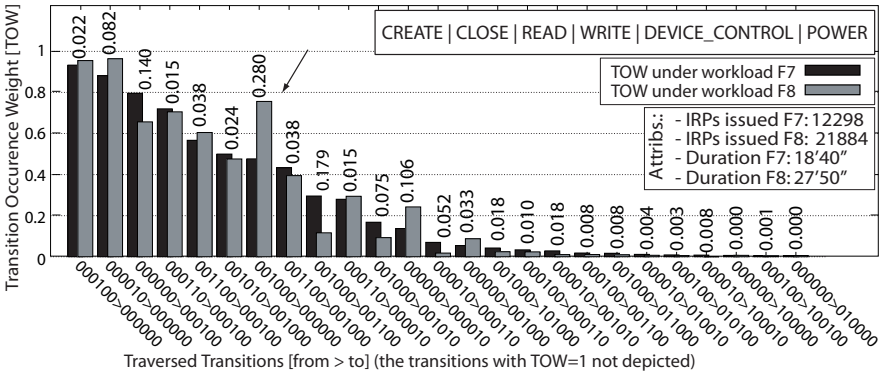

Figure 6.4: TOW comparison for F7 and F8

Even though the same modes and transitions were visited under both workloads, the distance between TOW values show a wider distribution than the MCW values. Figure 6.4 indicates the transition 001000→000000 (the bars pointed by the arrow in Figure 6.4) as having the largest difference between TOW values of F7 and F8, with 0.280 bias towards F8. This transition is traversed when the READ operation finishes and the DD returns to the *idle* mode. A closer inspection reveals that F7 compensates this difference with higher TOW values of the other transitions originating in 001000, i.e., transitions to the modes 001100, 001010 and 101000. As these modes are located on a lower level than 001000, this reveals the tendency of F7 to start the concurrent execution of WRITE, DEVICE_CONTROL or CREATE while still

running READ operations. This behavior might disclose potential problems related to concurrency handling or indicate places for optimizations.

Therefore, for comparing the effects of several workloads on a DD, the Euclidean distance between MCW values of each mode or between TOW values of each transition can be evaluated. Moreover, if the distance is smaller than a chosen threshold, then the compared workloads are considered equivalent. This constitutes feedback for ascertaining the adequacy of a testing campaign versus operational usage. The next section provides a detailed example on how the Euclidean distance can be used to analyze the relative similarity across workloads.

6.2.2 Many-to-many Comparison

This section presents a preliminary quantification method to ascertain the relative closeness between the execution pattern of a testing method and OPs generated by field workloads. This visually estimates the chances for the chosen testing strategy to cover the code areas that actually occur with a high likelihood as based on the mode and transition quantifiers presented in Chapter 5.

We use *multidimensional scaling* (MDS) plots to graphically display the distances between the workloads used to exercise the `flpy_XP` DD. MDS is a statistical technique used to visually express the degree of similarity (or dissimilarity) among objects belonging to a population, each object being characterized by a set of attributes. For each attribute a distance matrix is computed among the objects. To evaluate the similarity among the objects, the distance matrices associated with each object attribute are aggregated in a weighted average. The MDS plot is computed using this average. For the MDS plot depicted in Figure 6.5 we have used the Euclidean distance among the MCW values of the corresponding modes visited by each workload. Similarly, Figure 6.6 is computed using the Euclidean distance among the TOW values of the corresponding transitions for each workload. For instance, the closeness between two points in Figure 6.5 indicates that the associated workloads have visited the same modes, generating similar MCW values for each of them.

Figure 6.5 shows the MDS plot of the workloads F1 – F8, where the attributes of the objects are the MCW values of every sojourned modes of the OPs (i.e., 15 modes, see Table 5.2), with a $\lambda = 0.5$. If a workload did not visit a certain mode in our experiments, the MCW value of that attribute is zero. The MDS plot in Figure 6.5 reveals that the *DC2* is most similar to F4 and F5, two workloads that are representative for the operations performed when the floppy DD is loaded and unloaded from the OS. The comparison

between the modes of the F7 and F8 presented in Figure 6.3 is visually confirmed by the MDS plot, as the points associated with the two workloads are very close to each other.

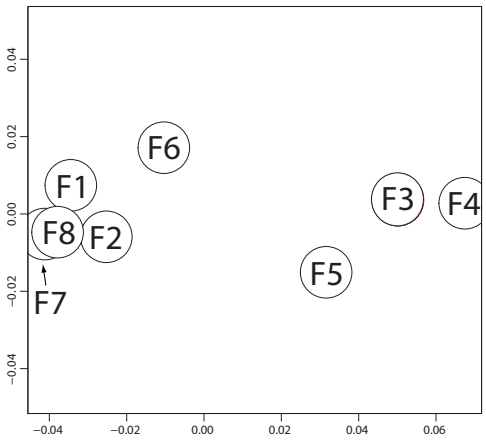

Figure 6.5: The distance among workloads (modes only)

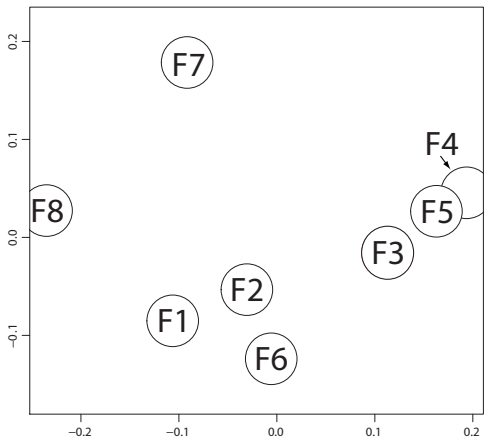

Figure 6.6: The distance among workloads (edges only)

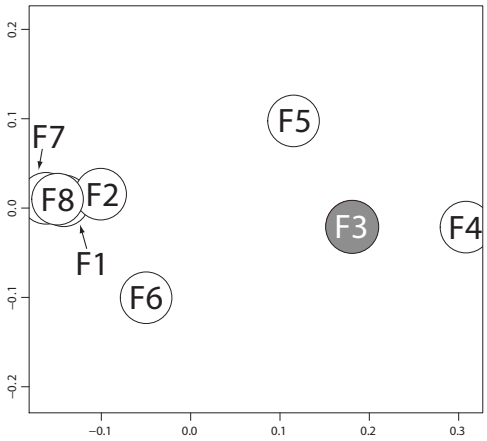

Figure 6.7: MDS plot considering two modes only

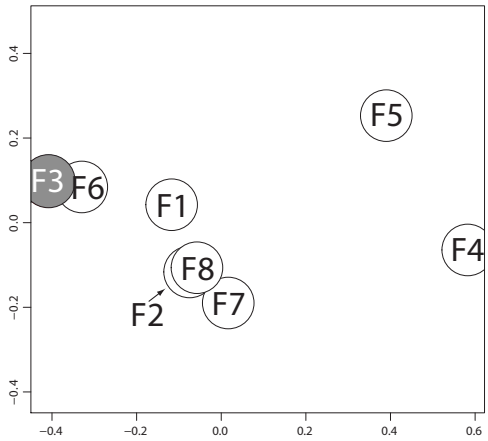

Figure 6.8: MDS plot considering two edges only

To examine how the workloads compare for the traversed edges, in Figure 6.6 we have considered as object attributes the TOW values of every transition of the OPs. Here, F7 and F8 are located farther away from the rest of the workloads. This effect is explained by the fact that F7 and F8 actually traverse all the 34 transitions that act as attributes of the objects in this plot, while the rest of the workloads traverse only a small subset of them.

As mentioned before, we assigned equal weights to modes and transitions for the MDS plots in figures 6.5 and 6.6. To accurately ascertain how close a testing campaign is from the manner the DD is exercised under realistic workloads, one should assign heavier weights to the modes and to the transitions carrying the most of interest when the inter-object distances are computed. For instance, Figure 6.7 shows the relative similarity among the workloads when only the initial mode (000000) and the mode associated with DEVICE_CONTROL (000010) are considered as object attributes, all the rest of the modes being ignored (all considered workloads visit these two modes). The MDS plot in Figure 6.8 is computed when weight is assigned only to the transitions between the same two modes.

While in Figure 6.7 the spatial configuration remains the same as in Figure 6.5 (despite the fact that F1, F2, F6 and F7 cluster closer to each other), the positions change dramatically in Figure 6.8 as compared to Figure 6.6. The closeness between F3 and F6 reveals that *DC2* generates an execution pattern which is very similar to the one recorded for F6. Therefore, using the DD state quantifiers introduced in Chapter 5, multidimensional scaling analysis can be successfully used on the data provided by our OPs to quantify the relative similarities among several workloads (for instance, field workloads vs. in-house testing workloads). This reveals new possibilities for statistical measurement of test coverage in terms of execution patterns.

Interestingly, our MDS plots revealed the tendency of *DC2* to cluster closer to the workloads that are associated with driver maintenance activities, F4 and F5. Given that *DC2* is a tool for testing the robustness and security of DDs, this tendency indicate that the *Enable* and *Disable* are exercising DD functionalities considered to be potentially harmful by Microsoft's OS developers. Moreover, it is reasonable that *DC2* shows dissimilarity with the other workloads, which are mostly associated with floppy disk DD's customary operations.

6.3 Test Space Reduction Tunability

To evaluate the test-space reduction enabled by our OP obtaining approach, we compared it with the total test space and the DD's OSS, considering both modes and transitions that need be covered by a testing campaign. At the end of this section Table 6.2 summarizes the overall improvement introduced by the current approach.

6.3.1 First-pass Reduction – The OSS

In figures 6.9 and 6.10, for each DD, we have selected the workload that exercised the largest set of modes and transitions, respectively. For instance, for the `flpy_XP` DD we selected the workload F7, as it issues IRPs belonging to five distinct types out of the six types supported by the respective DD. This indicates a *total state space* size of $2^6 = 64$ modes and $6 \cdot 2^6 = 384$ transitions. From this set, only 15 modes (23.44%) and 34 transitions (8.85%) were visited (see Table 5.2, columns 5–7). In figures 6.9 and 6.10, we call this early reduction step *first-pass reduction*.

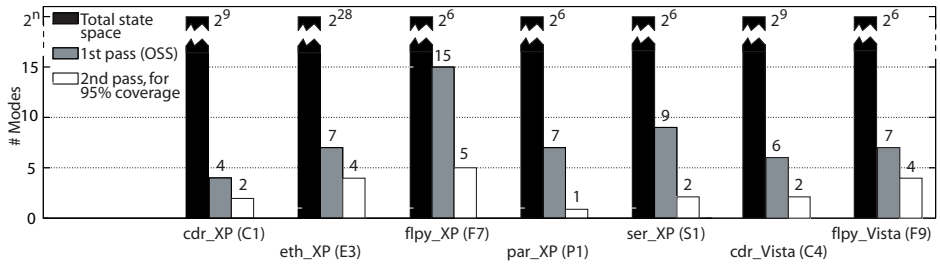

Figure 6.9: Test space reduction for the DD modes.

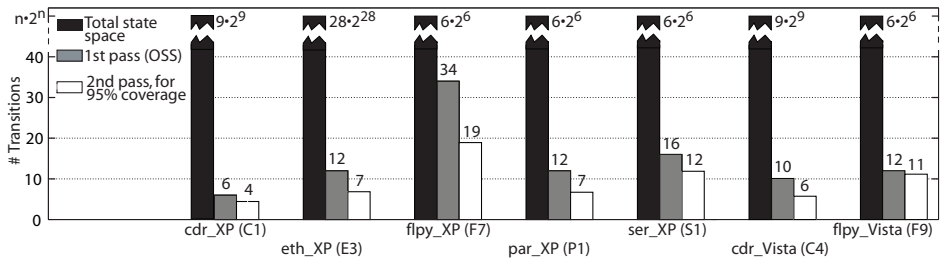

Figure 6.10: Test space reduction for the DD transitions.

While the first-pass reduction solely divides the total state space into two classes (visited and non-visited), we additionally utilize the newly introduced execution quantifiers for modes and for transitions. They permit a finer differentiation among the visited modes (and among the traversed transitions), offering subsequent testing campaigns a richer insight about the modes and transitions that need consideration. In figures 6.9 and 6.10 this step is called *second-pass reduction* and it is based on the relative ranking among modes and transitions, respectively. The process of obtaining such rankings is discussed in Section 6.1 and depicted by Figure 6.1.

6.3.2 Second-pass Reduction via Priority Ranking

To explain the mechanism of the second-pass reduction, we introduce a threshold T specifying the desired test coverage level. Onwards, the word "*coverage*" refers to the coverage of the modes and transitions in our model. In relation with the ranking of modes (or transitions, respectively), T gives the reduction of the second-pass. For example, if a test coverage of 95% is desired for the visited modes under the workload F7, then they are selected from a list ranked by their MCW weights. Figure 6.11 depicts the cumulative MCW value of the modes visited under F7. The modes are ordered in decreasing MCW order, from left to right. Therefore, when the example coverage goal is 95% of the visited modes, the first five leftmost modes (marked in the figure) are selected. This gives a test space reduction of 66.66% compared to the OSS presented in Chapter 4. When a stricter 99% coverage of visited modes is desired, three more modes are selected, still giving a 46.66% reduction relative to the first-pass.

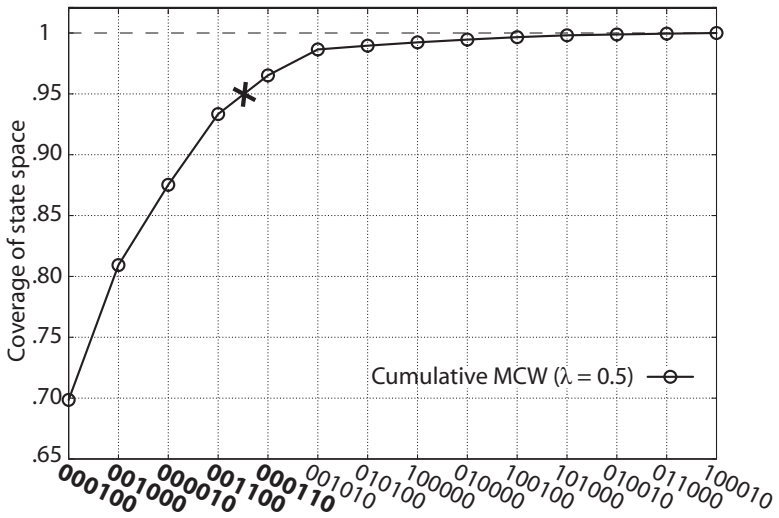

Figure 6.11: The cumulative coverage of the DD's modes for F7. The modes are ordered in decreasing MCW order from left to right.

Figure 6.11 indicates that the most visits are concentrated to a very small number of modes. This trend also holds for the transitions, indicating a high reduction in modes and transitions selected by the second-pass, also when the desired coverage is high. Therefore, this mechanism of selecting the modes of interest based on the rankings given by the execution quantifiers can be tuned using the threshold T to best fit coverage goals. Figures 6.9 and 6.10 depict the second-pass reduction of modes (and transitions, respectively) for

95% testing coverage. Table 6.2 lists the reduction for each of the workloads considered in figures 6.9 and 6.10.

Table 6.2: Test space reductions of the OP relative to the OSS. (for a coverage threshold $T = 95\%$)

Quantifier	**C1** [%]	**E3** [%]	**F7** [%]	**P1** [%]	**S1** [%]	**C4** [%]	**F9** [%]
MCW	50.00	42.85	66.66	85.71	77.77	66.66	42.86
TOW	33.33	41.66	44.11	41.66	25.00	40.00	8.33

Hence, when test resources are limited, we believe that the testing effort can efficiently be tuned for the desired coverage level in order to first cover the modes and transitions associated with high likelihoods to be reached in the field. Assuming that the test effort is equally distributed among the visited modes, this result indicates that a significant reduction in the amount of testing is possible, without affecting its adequacy. While the test case creation and prioritization are out of the scope of this thesis, we believe that existing test methods can directly utilize the test guidance insights offered by our profiling methodology.

6.4 Experimental Aspects

In this section we discuss different issues related to our experimental procedures. We start by describing the threats to the validity of the obtained results, then we discuss the overheads of the used monitoring mechanisms. This section also addresses the efforts required to obtain OPs and concludes by presenting the lessons learned.

6.4.1 Threats to Validity

We discuss here the issues that can potentially limit the generalization of the experimental profiling approach for a wider population of DDs. The DDs chosen for our experiments are WDM-compliant DDs and therefore do not represent a random selection, as they were chosen from the set of DDs installed on the host machines. However, we believe that they are representative for the population of Windows XP DDs as the WDM DDs constitute the main type of DDs available for this OS [Oney, 2003].

External Threats to Validity

The external threat to validity is related to the generalization of the experimental results to all currently existing DD types. To address this issue we selected as diverse DDs as possible (see Table 5.2). Also, the filter driver used to monitor the selected DD's runtime activity was kept general enough to be ported on virtually any WDM-compliant DD. The representativeness of the workloads chosen to exercise the DDs does not impact the validity of the experiments presented in this thesis as we are not attempting to completely profile the field behavior of the DDs. Our purpose here is to propose a general methodology to quantify DD operational behavior.

Instead, the workloads listed in Table 5.2 were selected as they trigger a wide set of specified DD functionalities. However, for obtaining accurate OPs we recommend collecting them in the field (e.g., during a beta-testing campaign involving many users); when this is not possible, a wide selection of workloads for the target DD has to be used to ensure accuracy of the OP quantifiers.

The *OP-Builder*, the software tool we created for offline log parsing and analysis was found not to easily scale to extremely large log files[1]. This limitation is due to an inefficient object management of the JVM which leads to memory leaks. However, this has no impact on the results presented in this thesis but we are aware that the re-usage of our tools in larger DD profiling projects may require handling this issue, possible by running it in other (better) JVMs or, ultimately, porting it to other programming language.

Internal Threats to Validity

The internal threats to validity of the presented experimental results represent conditions influencing the dependent variables of the experiments. The main threats to internal validity are represented in our experiments by *confounding* and *instrumentation*.

Confounding occurs whenever an extraneous variable changes along with the independent variable, preventing the inference of a causal relationship between the independent and dependent variables. To avoid this threat to the validity of our experiments, we have carefully ensured that no other active process accessed the target DD in the experiment's time window. To detect and filter out the I/O calls not belonging to the considered workload – DD pair before proceeding with OP analysis, we accordingly marked the messages intended for logging.

[1]Some log files have hundreds of thousands entries.

Instrumentation threat is produced by changes introduced to the studied system by the measurement instrument itself. To reduce this internal threat, we have restarted each DD (the DD's code was removed and then reinstalled in the OS kernel) after each experimental run. Also, to keep our approach as non-intrusive as possible, only the filter driver was inserted into the target OS kernel on the I/O requests' path. With its code written with efficiency in mind, the filter driver only captures, logs and then forwards the IRP traffic. The next section presents the overheads introduced by our DD monitoring mechanisms.

6.4.2 Monitoring Overhead

Table 6.3 lists the average overheads introduced by the filter driver for the DDs presented in this study. Each DD was exercised with the *DC2* testing tool from Microsoft. It was run 64 times with, and then 64 times without the filter driver installed in the driver stack.

Table 6.3: The overheads introduced by the filter driver in terms of *elapsed time* and *CPU time* for all the studied DDs. The data represents averages over 128 runs of each DD, 64 with and 64 without the filter driver installed.

Driver (*see Table 5.1*)	**OS**	**Elapsed Time**			**CPU Time**		
		w/o filter t_0 [ms]	**w/ filter** t_1 [ms]	**Increase** $\frac{t_1-t_0}{t_0}\cdot 100$ [%]	**w/o filter** t_2 [ms]	**w/ filter** t_3 [ms]	**Increase** $\frac{t_3-t_2}{t_2}\cdot 100$ [%]
`cdrom.sys`	XP	5418	5428	**+0.18**	276	340	**+23.18**
`cdrom.sys`	Vista	1600	1685	**+5.31**	120	140	**+16.67**
`flpydisk.sys`	XP	3010	3087	**+2.55**	165	208	**+26.06**
`flpydisk.sys`	Vista	3070	3308	**+7.75**	234	289	**+23.50**
`parport.sys`	XP	4896	4929	**+0.67**	205	253	**+23.41**
`serial.sys`	XP	1230	1238	**+0.65**	86	98	**+13.95**
`sisnic.sys`	XP	3516	3585	**+1.96**	186	236	**+26.88**
AVERAGE		3248.5	3322.6	**+2.72**	181.7	223.4	**+21.95**

We measured two temporal parameters on each run: (a) *elapsed time* and (b) *CPU time*. The elapsed time represents the total duration of a run (i.e., calculated from the moment when the process starts until it finishes), while the CPU time is the total time spent by the *DC2* process in the CPU (i.e., executing computing-intensive operations).

The results presented in Table 6.3 show that our filter driver induces a minimal overhead in terms of elapsed time averaging 2.7% increase, despite the fact that the CPU time increase is on average almost 22% higher than without our filter driver installed.

However, this apparently high impact is actually insignificant as our workload is mostly I/O intensive, and the CPU time represents only 6% of the total elapsed time. The data presented in Table 6.3 also show that the increase is similar across all the studied DDs, indicating the fact that the overhead introduced by our filter driver is stable irrespective of the type of the DD that it is installed upon. Interestingly, for the DDs profiled under Windows Vista the influence of the filter driver was higher than for the corresponding Windows XP DDs.

6.4.3 Efforts for Obtaining Operational Profiles

Development Effort

The primary effort in obtaining the experimental data presented in Chapter 5 was spent in implementing the filter driver and the *OP-Builder* tool (see Figure 5.3). Due to the constraints imposed by WDM and DDK [Oney, 2003], the filter driver was programmed using plain C. In contrast, *OP-Builder* was programmed using Java.

The SLOC (source lines of code) metric gives an insight on the effort required to build the tools. SLOC counts the number of physical lines of code, whereas the blank and comment lines are not counted (for a detailed definition see Wheeler [2001]). Hence, the filter driver has a SLOC count of 378, while the *OP-Builder* has a SLOC count of 1317. The required amount of work estimates to approximatively one person–month for the filter driver and a little over four person–months for developing the *OP-Builder* tool (we have used the COCOMO metric [Boehm, 1981] to estimate the development cost).

Usage Effort

As the tools developed for monitoring and analyzing purposes are supposed to be widely portable across the population of WDM-compliant DDs as possible, we expect that their re-usage effort is kept to a minimum. No re-compilation of the tools is required for analyzing each DD, the configuration parameters are implemented as editable text files.

Hence, the usage effort is represented only by tool configuration. In the case of the filter driver, the name of the target-DD has to be specified in a configuration file, while for the *OP-Builder* tool one must list the supported IRP calls in a text file.

Fifteen students were asked to run profiling sessions for several DDs, after they were instructed how to use the tools. Thanks to the high degree of

automation and simplicity of the configuration mechanisms, all students were able to successfully produce OPs for the given set of DDs in a few minutes of work. Table 5.3 contains, in part, the results of their work.

Analysis Effort

The *OP-Builder* tool takes log files containing I/O traffic information as inputs and produces OP graphs as outputs. The OP graphs are represented in DOT language, thus they can be viewed using Graphviz [Simionato, 2004] or other open-source graph visualization tools. The analysis process is extremely fast, the *OP-Builder* tool can process log files at an average speed of 25000 IRPs per second. The longest time required by the log-analysis stage in our experiments was around three seconds, namely for the workload C1 (see Table 5.2).

6.4.4 Experimental Issues and Lessons Learned

Our experiments showed that some workloads issue large amounts of I/O requests per time unit, an example being the *DC2* workload. This puts high pressure on the monitoring mechanisms, introducing the risk that important behavioral information may be lost. This happens due to the *DebugView* kernel debugger (see Figure 5.3) which is unable to track all messages when the arrival rate is extremely high.

A discussion with the developers of the *DebugView* revealed that the messages are stored in a temporary buffer. When the buffer is full, new messages are dropped. We are currently working on an improved version of the logging mechanism which circumvents this problem by directly logging into a selected file.

Our filter driver has been built to carefully ensure that no I/O requests are lost, and we have verified offline all traces to ensure that no invalid transitions are made (two bits in the mode are changed) and that all IRP pairs (incoming and outgoing) are matched. Note that the results presented in Table 5.2 are complete with respect to the tracked I/O requests.

Summarizing the experiences gained from our experimental efforts, Table 6.4 represents a collection of suggestions intended to further improve the testing of DDs by applying the OP introduced quantifiers.

6.5 Chapter Summary

This chapter introduced the usage of the DD operational profiles for test support activities. First, the mode quantifiers were used to create rankings

Table 6.4: Suggestions for proposed metric usage toward improved DD test campaigns.

► **Filter driver:** build a filter driver (or reuse ours) to monitor the I/O traffic between I/O Manager and the DD of interest; keep it as simple as possible, to avoid negative impact on system performance and reliability.
► **Quantifiers:** verify the log-files generated by the filter driver for lost IRPs to ensure the validity of the obtained data; using the validated logs, compute the OP quantifiers (as presented in Section 5.2) for each mode and transition of the OP.
► **Choosing λ:** λ is a coefficient designed to enable tuning the test priority toward often visited modes or toward modes where the DD spent large time amounts; vary λ according to test interest and use the obtained MCW values to prioritize the test campaign.
► **Seldomly sojourned modes:** the modes with small values of MOW or MTW should not be overlooked in testing, as their activities are usually crucial for DD's service provision, being associated with management of the DD.
► **Find patterns:** carefully inspect the execution pattern to identify I/O sequences relevant for DD testing (and build test cases accordingly).
► **Compare workloads:** to express the relative similarity among several workloads, measure the Euclidean distance of the mode and transition weights and define a equivalence threshold; for weights smaller than the threshold, the workloads are said to be equivalent.
► **Capture field workloads:** when possible, capture the DD's OP in the field and compare it with the workload used in-house for testing to improve the adequacy of the test process.

of the OP modes, thus alleviating the test prioritization effort. Second, two methodologies were presented that permit one-to-one and many-to-many cross comparisons among workloads. These methods are useful as they enable the choice of test workload as similar as possible to the field workload. Third, the effectiveness of the introduced DD OPs was analyzed from test reduction perspective as a two-pass process. While the first-pass is represented by the transition from the total state space to the OSS, the second-pass is a tunable step further and significantly reducing the test space that need be covered without losing adequacy.

The chapter outlines the key aspects of the experimental process, the presented discussion including the threats to validity and the costs required to construct DD OPs both in terms of monitoring overhead and also as implementation effort. The chapter concludes with a summary of the lessons learned in our experimental endeavor towards constructing DD OPs.

Chapter 7

Execution Path Profiles

How can the code-paths taken at runtime be inferred without access to the driver's source code? Can hotspots in the driver's code be highlighted to help prioritize testing accordingly?

The methodology for obtaining DD operational mode profiles presented in the previous chapters is exclusively based on monitoring the I/O call interface. By capturing the I/O request traffic at this level we first defined the DD state and its *total state space* (Chapter 3), then identified its *operational state space* (Chapter 4), and finally highlighting its *operational profile* (Chapter 5).

In this chapter we now present experimental results obtained by investigating the DD's *functional interface* to the OS kernel, in addition to the I/O call interface. At the functional interface of the DD, the calls made to driver-external functions (located in kernel libraries) can be ascertained. While the information obtained at the I/O call interface permits only inter-mode profiling of the DD's activity, the flow obtained from the functional interface allows for a deeper insight into the DD's runtime behavior, as they reveal the code paths followed in the operational phase.

The experimental evaluation of the code paths show a key phenomenon, namely the tendency of call traces to cluster with respect to the source code being executed. Consequently, we present a cluster analysis method to assess the relative similarity of the executed code paths. The obtained trace clusters represent (together with their occurrence indexes) effective representations of a DD's execution hotspots. From a testing perspective, this strongly indicates the possibility to significantly reduce the testing effort needed to cover the exercised code paths by thoroughly testing only a single representative code

path from each equivalence class. These represent the thesis contributions **C1** and **C5** (see Section 1.2.2).

Additionally, we show how the number of equivalence classes can be decided by varying the similarity threshold (the *cutoff* factor of the *dendrogram* – a tree-like structure describing the clustering). This represents a powerful tool for directing the efforts that a subsequent testing campaign needs to undergo. As a case study, both conceptual contributions are experimentally evaluated against an actual Microsoft Windows DD.

7.1 Code Tracing – A Basis for Execution Profiling

Execution profiling information is an important prerequisite for helping DD validation. It is an abstract model to describe how a DD behaves under the influence of external stimuli. As such it can help DD testers identify which part of the DD code is exercised for a representative workload. This can be used to guide selection of test cases by focusing on the *most frequently used parts in an operational setting*, which may substantially differ from statically selected test cases.

In this chapter we develop a profiling methodology for kernel-mode DD execution paths by considering an additional communication interface alongside with the I/O requests considered in the previous chapters of this thesis (chapters 3 – 5). According to this communication paradigm, at runtime a DD acts as a consumer of the services (i.e., *functions*) provided by various OS kernel libraries.

Therefore, a DD's runtime activity can be defined by the sequences of calls made to external functions. As DDs act on kernel calls, the call sequences are delimited by the I/O requests generated by the OS, and thus infer the execution path taken in the DD's code. This helps to evaluate and to compare the effects of different workloads (i.e., test suites or individual test cases) by revealing execution hotspots. The presented process for caption and evaluation of the call traces does not require source code access to any of the involved OS components.

Overall, this section outlines a methodology to obtain execution profiles for kernel DDs. By using tracing information from two DD communication interfaces, our technique provides insights that help better understand a DD's runtime behavior in terms of execution paths.

A significant amount of research was dedicated to tracing code execution but very little of it is applicable to black-box level SW, and more precisely to kernel-mode DDs. Johansson et al. proposed a selection method for *SWIFI* injection triggers which is based on call blocks of DD-external functions [Johansson et al., 2007a]. The methodology presented in this chapter for profile construction is similar in terms of the used monitoring strategy, but in contrast we consider the effects of the kernel's I/O requests on the DD's behavior.

Mendonca and Neves used a *SWIFI* technique to evaluate the robustness of the kernel libraries [Mendonca and Neves, 2007]. The target functions were selected statically by inspecting the import tables of the DDs of several Windows installations and choosing the ones that are used by most of the

DDs. In accordance with Weyuker's recommendations [Weyuker, 2003], our results suggest that the target functions should be selected on a dynamic basis (using profiling), by using occurrence indexes to guide the selection process.

Ball and Larus acknowledged the application of path profiling for test coverage assessment, "*by profiling a program and reporting unexecuted statements or control flow*" [Ball and Larus, 1996]. They used binary instrumentation to obtain instruction traces that reveal a program's control-flow to identify paths and their execution frequencies. The paths ended at loop and procedure boundaries. An extension is represented by the "*whole program paths*" described in [Larus, 1999], which crosses both boundaries to reveal a better picture of a program's execution patterns. Though, these approaches are not directly applicable to DD as the they are implemented as function libraries rather than programs in the classical sense. Moreover, instrumentation induces a high execution overhead and produces large amounts of data, two characteristics which penalize the use of this approach inside the OS kernel space.

Leon and Podgurski used profiles generated by individual test cases and a clustering technique for evaluating test suite minimization by selecting one test case per cluster [Leon and Podgurski, 2003]. The profiles used were generated by third-party tools, so the cluster analysis had to rely on their accuracy. While test cost reduction is out of the scope of this thesis, we focus on building viable and accurate DD profiles, as a prerequisite mean to reducing test efforts.

7.1.1 The PE/COFF Executable Format and DLL-Proxying

The communication between OS kernel and DDs is not limited to the I/O request scheme. A DD also communicates with the OS kernel using a second interface, which we onwards call the "*functional interface*" (detailed in Section 3.2.2). Enabled by the concept of dynamic linking, at this communication level the parties involved are kernel libraries and DDs, as image files. In fact, this scheme forms the basis of OS modularity, and is the most commonly used data communication paradigm between binaries.

The OS provides a set of kernel libraries containing functions required by the different kernel components. Each library publishes a list of the available functions. On the other side, the DDs (as consumers of the services provided by the libraries) contain a list of necessary external libraries and for each of them a list of the used functions from the respective library. For both kernel

libraries and DDs the lists mentioned above are stored in the headers of the binary files. In Windows, the PE/COFF format [Microsoft Corp., 2008] specifies the file headers that permit a Windows executable file to publish the contained functions and variables (*exports*) and to use functions defined externally by another library (*imports*).

Figure 7.1 depicts a DD importing functions implemented in two external kernel libraries, *Lib1.sys* and *Lib2.dll*. Each contains an Export Address Table (EAT) that publishes a list of functions exported by the respective library. At runtime, the DD links to the kernel libraries on demand, when the result of the functions *"foo"* and, respectively, *"bar"* are needed. Therefore, the header of the DD file contains an Import Address Table (IAT) for each of the needed libraries. The IAT contains only the function names which are used in the DD's code.

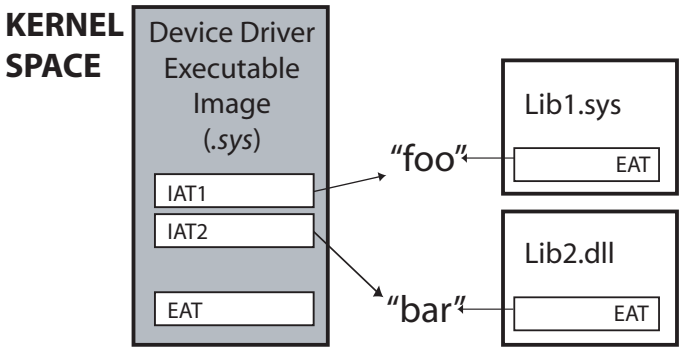

Figure 7.1: A DD importing functions from two libraries.

At DD load time, the OS automatically checks if all the required libraries are present in the system by inspecting the DD's IATs. If they cannot be found, an error message is issued and the DD loading is aborted. At load time, no verification is done to check if the found libraries actually contain the necessary functions for the DD to execute correctly. Only at runtime, when parts of code containing calls to external functions are reached, the DD accesses the associated library.

The work presented in this chapter relies on the ability to capture the calls to external functions at DD runtime. While various methods for capturing calls to externally located functions exist (eg., *Detours* [Hunt and Brubacher, 1999] or *Spike* [Vasudevan and Yerraballi, 2006]), they are specific to user-space software and are therefore not directly applicable to kernel-mode programs. In contrast, we utilize a kernel space mechanism to monitor the function calls.

Therefore, we have chosen to implement a *DLL-proxying* technique. Briefly, DLL-proxying consists of building a DLL library acting as a wrapper of the original library. In order to leave the functionality of the DD unaffected, the wrapper library has to implement all the functions required by the DD, or to forward its calls to the original library. By modifying the IAT tables of the target DD to point to the wrapper library instead of the original one, the wrapper library (termed as *DLL-proxy*) is interposed between the two parties. Section 7.3 details our implementation of DLL-proxies inside the Windows OS kernel.

Our kernel-mode library wrappers are used exclusively for capturing the sequences of functions called by a DD at runtime, when exercised by a selected workload. Consequently, we only need to log the function names but not modify any parameters or behavior of the wrapped kernel APIs. Therefore, the overhead induced by the DLL-proxy is kept to minimum.

7.1.2 Call Strings as Code Path Abstractions

As external function calls correspond to DD code being executed as a result of I/O requests (or other OS kernel maintenance requests), grouping them using I/O requests as boundaries is intuitive. Therefore, we introduce the notion of *call string* as follows:

Definition 15 (**Call String**). *A* call string *(CS) is a sequence of DD-external function calls issued at runtime by a DD, delimited by incoming and outgoing I/O requests.*

We consider each CS an abstraction representing the *code path* taken by the DD at execution time. As we use the incoming and outgoing I/O requests as CS delimiters, each CS can be associated with a DD mode and, subsequently, with an I/O request dispatch function.

Accordingly, the *execution path profile* can be now defined as follows:

Definition 16 (**Execution Path Profile**). *The* execution path profile *(EPP) of a DD is the complete set of the call strings captured at runtime in the time interval spanned by the execution of a workload exercising the respective DD.*

Illustrating the CS capturing method, the left part of Figure 7.2 shows an abstract representation of the WDM-compliant DD's source code with dispatch routines for handling READ and WRITE requests. Assuming that the DD can handle only those two I/O requests, the visited modes are defined by bit strings with length two; the first bit is associated with READ and the

second with the WRITE operation. Note that both dispatch routines call functions implemented externally by other kernel libraries.

Assuming that at a certain instant the DD receives the READ request followed by an WRITE request, the log file storing the events obtained by monitoring the two communication interfaces is depicted on the right side of Figure 7.2. Hence, the call strings CS_i and CS_{i+1} can be constructed and associated with the DD modes 10 and, respectively, 01.

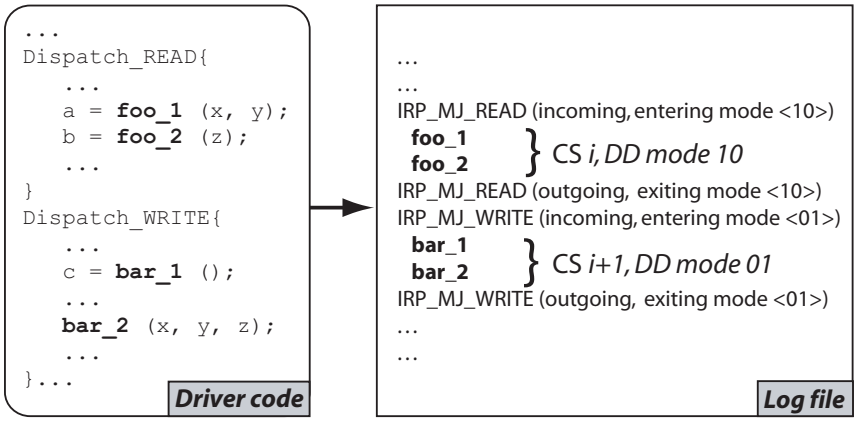

Figure 7.2: The code path taken in a DD when READ and WRITE requests are called.

Consequently, the EPP can be studied from two perspectives: (a) *per mode basis*, i.e., CSs belonging to the same DD mode are compared to reveal possible differences in the code paths taken each time the DD performs the activity associated with the respective mode, and (b) *per CS basis*, i.e., all CSs are compared among themselves to identify similarities and to group them accordingly in equivalence classes. Hence, we define the term *execution hotspot* as follows:

Definition 17 (**Execution Hotspot**). *A group of similar CSs belonging to the same equivalence class represents an* execution hotspot*. The magnitude of each hotspot is given by the occurrence index of the CSs contained within the equivalence class.*

The methodology for building EPPs presented in this chapter reveals the execution hotspots together with their magnitudes. The construction of the equivalence classes is achieved by employing a cluster analysis algorithm, as described in the following section.

From the testing perspective, our DD code profiling approach can be used by subsequent testing campaigns as it reveals the code paths taken by

a black-box DD-under-test (for instance when a test case is run against it). Moreover, a testing campaign can be prioritized by associating a high priority to the CSs appearing often in the captured logs. Therefore, DD testing might benefit by using the information contained in the CSs to avoid testing code paths which are never taken in the field and balance the efforts towards code areas having a high likelihood to be reached in the field.

7.2 Clustering as Execution Hotspot Identification

Given the large amount of data collected in the monitoring phase, a data clustering method applied to the EPP greatly facilitates organizing and interpreting the data trends[1]. *Cluster analysis* is a multivariate technique that helps partitioning a population of objects into equivalence classes.

The partitioning decision is taken on object similarity, i.e., similar objects are grouped together in the same *cluster*. The most common clustering approaches are hierarchical and partitional. Usually slower than hierarchical algorithms, *partitional clustering* initially divides (randomly) the object population into k clusters, improving the clusters at each step by redistributing the objects. *Hierarchical clustering* approaches fall in two classes, agglomerative and divisive.

Agglomerative clustering (also called bottom-up clustering) initially assigns each object into its own cluster, at each step similar clusters are merged. The agglomerative clustering algorithms stop when all objects are placed in a single cluster, or when a number of k clusters (given as a parameter to the algorithm) remain. *Divisive* clustering (top-down clustering) algorithms initially assign all the objects from a given population to a single cluster, divided at every step in two non-empty clusters. A divisive clustering algorithm stops when each object sits in an own cluster or when a number of k clusters is reached.

We use automated agglomerative analysis to divide the EPP into similar clusters. We use *AgNes*, an agglomerative algorithm provided by the *R* statistical programming environment [Ihaka and Gentleman, 1996]. *AgNes* requires as input a matrix containing the distances between every pair of objects, in our case CSs. It outputs a *dendrogram*, which is a tree-like representation of the clustering.

The figures 7.8 and 7.9 represent examples of such dendrograms. The

[1] A log file contains many call blocks, hence a manual analysis thereof being impractical. For instance, *Sandra* produced over 9500 call blocks, and *DC2* more than 5000.

CSs are represented as leaves, and branches intersect at a height equal to the dissimilarity among the children. Cutting the dendrogram at a given height reveals the clusters and the contained call sequences at the respective distance. That is, the *similarity cutoff* of the dendrogram indicates the equivalence classes that partition the CS population for the respective distance. For a cutoff set at 0, the equivalence classes contain only the CSs which are identical. Therefore, the cutoff value acts as a tunable mask for CS diversity.

7.2.1 Metrics to Express Call String Similarity

To obtain relevant dendrograms of the CS clusters, an appropriate similarity metric has to be selected. In the areas of bio-informatics and record linkage (duplicate detection) researchers have developed a series of metrics to quantify the relation between two strings. Depending on their application area, some metrics express the similarity while other measure the difference (dissimilarity) of the compared strings.

The *Levenshtein* distance (d_L) is based on the edit distance between the compared strings [Levenshtein, 1966]. Given two strings s_1 and s_2 whose distance is to be computed, Levenshtein distance express the minimum number of operations needed to transform s_1 in s_2 or viceversa. The considered operations are character *insert*, *delete* or *substitution* and they all have the cost of 1.

Used in bio-informatics to decide global or local alignments for protein sequences, *Needleman-Wunsch* [Needleman and Wunsch, 1970] and *Smith-Waterman* [Smith and Waterman, 1981] distances are versions of the Levenshtein metric, additionally considering gap penalties (a *gap* is a subsequence that does not match).

Jaro distance is not based on the edit distance, but instead uses the amount and order of the common characters [Jaro, 1989]. The Jaro distance is expressed by the following formula:

$$d_J = \frac{1}{3}\left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}\right) \tag{7.1}$$

where m is the number of matching characters and t is the number of necessary transpositions. Two characters are considered matching if they are not farther than $\left\lfloor \frac{max(|s_1|,|s_2|)}{2} \right\rfloor - 1$ from each other. An extension of the Jaro distance was proposed by Winkler, in order to reward with higher scores the strings that match from the beginning (they share a common prefix).

Therefore, the *Jaro-Winkler* distance is defined by the formula

$$d_{JW} = d_J + [0.1 \cdot l(1 - d_J)] \tag{7.2}$$

where l is the length of the common prefix and d_J is the Jaro distance between the strings [Winkler, 1999].

Many other distance metrics exist and have been evaluated for various applications [Cohen et al., 2003]. We have also investigated several of them and subsequently chosen the Levenshtein and Jaro-Winkler metrics, as we believe they express best the distance among the CSs. Levenshtein was selected as it neutrally captures the variability of the CSs. As we expect the CSs to contain short, repetitive subsequences (generated by loops in the code path) and common sequences (generated by shared helper functions), we have also selected the Jaro-Winkler metric as it favors similarities between CSs showing this behavior.

To balance their effects and to minimize the impact of the metric choice on the final cluster structures, we combined them in a compound measure, a simple weighted average:

$$d_C = \frac{norm(d_L) + norm(d_{JW})}{2} \tag{7.3}$$

Our compound metric uses normalized values for both Levenshtein and Jaro-Winkler functions, therefore $0 \leq d_C \leq 1$. Being a dissimilarity function, small values of d_C indicate high similarity between the compared CSs. The distance matrix required by *AgNes* was computed using d_C for expressing the distance among every CS pairs.

7.2.2 Cluster Linkage Methods

Besides the distance matrix, *AgNes* requires that a clustering method is specified. *Simple linkage* merges at every step two clusters whose merger has the smallest diameter. This method has as disadvantage a tendency to form long cluster chains (i.e., at every step a single element is added to an existing cluster). *Complete linkage* merges clusters whose closest member objects have the smallest distance. This linkage method creates tighter clusters but is sensitive to outliers. To alleviate the disadvantages of simple and complete clusterings, *average linkage* groups clusters whose average distance between all pairs of objects is minimal.

AgNes provides a standard measure to express the strength of the clustering found in the population of CSs. A strong clustering tendency means larger inter-cluster dissimilarities and smaller intra-cluster dissimilarities. If

$d(i)$ is the dissimilarity of object i to the first cluster it is merged with divided by the dissimilarity of the last merger, the *agglomeration coefficient (AC)* is expressed by *AgNes* as the average of all $1 - d(i)$. With $0 \leq AC \leq 1$, larger AC values indicate a good cluster structure of the object population.

For our clustering analysis experiments presented in the next section we have used the average linkage method as we believe this choice factors out best the impact of CS distance variance among the object population.

7.3 Experimental Evaluation

For a comprehensive evaluation of the dual-interface DD profiling method presented in this section for obtaining DD EPPs, we have used it against the *flpydisk.sys* (v5.1.2600.2180), the floppy disk DD provided by Windows XPSP2.

Figure 7.3 depicts our experimental setup. To capture the requests occurring on the I/O call interface of the target DD we have built a filter driver and installed it between the monitored DD and the I/O Manager. The filter driver receives the incoming and outgoing I/O requests, logs them to a file and forwards them to the original recipient. As the filter driver does not rely on the implementation details of the underlying DD, it can be used to monitor virtually any WDM-compliant DD, as shown in practice by the experimental work in Chapter 5.

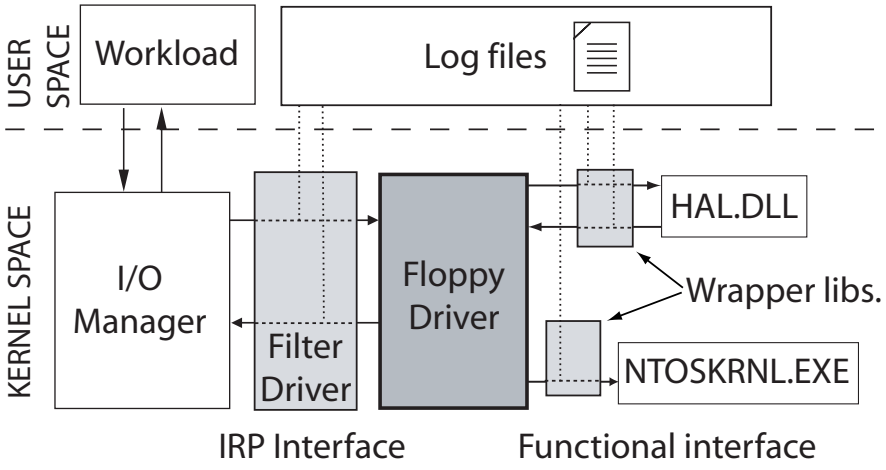

Figure 7.3: Obtaining the code paths taken at runtime.

The monitoring of the functional interface is more complex than that of the I/O call interface, as it requires building a wrapper library for each of the

kernel libraries imported by the floppy DD (Figure 7.3). *flpydisk.sys* imports functions from two kernel libraries, *NTOSKRNL.EXE* (61 functions) and *HAL.DLL* (4 functions). After building the library wrappers, the IAT tables of the target DDs were modified in order to look for the wrappers instead of the original libraries.

```
...
NTSTATUS
FASTCALL
WrapperIofCallDriver(
     IN PDEVICE_OBJECT DeviceObject,
     IN OUT PIRP Irp)
{
   PrintOut("IofCallDriver");
   return IofCallDriver(DeviceObject, Irp);
}
...
```

Wrapper code

Figure 7.4: A wrapper for the *NTOSKRNL::IofCallDriver* API.

Each API wrapper was built using exclusively the function prototypes provided in the header files available publicly from Windows DDK package. Each time the DD called a function, the API wrapper is called instead of the original function. The API wrappers are designed as extremely simple plain C constructs in order to minimize the computational overhead. When a wrapper is called, the call is logged and the call parameters are forwarded unchanged to the original function from the original library, as depicted by the code snippet in Figure 7.4. In this figure, *IofCallDriver* is the original function implemented by *NTOSKRNL.EXE* and *WrapperIofCallDriver* is our wrapper.

After the floppy DD is exercised by a relevant workload, the resulted log files are analyzed offline by a software application that extracts the CSs, builds the EPP and constructs distance matrix files. These files are then fed to the *AgNes* algorithm which builds clusterings of the CSs.

More precisely, the procedure followed to build the clusterings that evaluate the CS relative similarity is depicted in Figure 7.5 in a step-by-step manner: (1) collect the CSs by using the monitoring logs (EPP); (2) encode each function call to an Unicode character to be able to apply the string metrics; (3) calculate the distance matrix containing the distances between all pairs of CSs; (4) select the distinct CSs and count for each one the occurrence index; (5) construct a clustering from all distinct CSs to evaluate

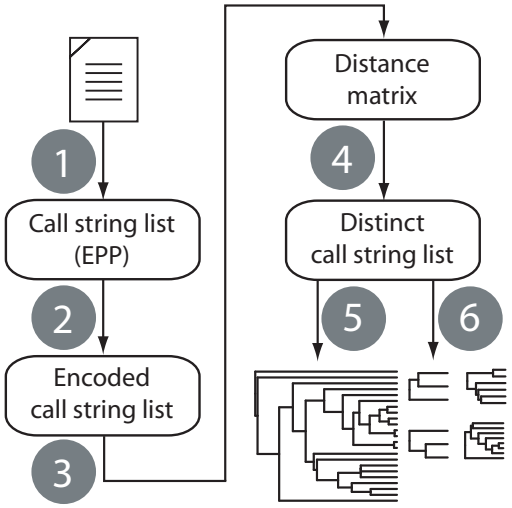

Figure 7.5: Our cluster analysis process.

inter-CS similarities; (6) for each mode, construct a clustering of CSs to reveal intra-mode paths.

Table 7.1: The workloads utilized to exercise the floppy DD and the overall experimental outcomes.

	#Called Imports				#CSs			
Benchmarks for flpydisk.sys	**Total**	**NT**[1]	**HAL**[2]	**#Modes**	**Total**	**Distinct**	**AC**	**Description**
`Sandra`	27	25	2	3	9545	51	.859	Performance benchmark
`DiskTestPro`	28	26	2	5	588	13	.735	Surface scan, format
`BurnInTest`	21	19	2	5	1438	24	.823	Reliability benchmark
`Enable_Disable`	42	38	4	3	136	10	.388	DD load and unload
`DC2`	21	19	2	4	5102	9	.644	Robustness benchmark

To exercise the DD properly, we have used commercial performance and stability benchmark applications designed for testing the floppy disk drive. We have also used a robustness testing tool, *DC2* (Device Path Exerciser). *DC2* is part of the DDK package and evaluates if a DD submitted for certification with Windows is reliable enough for mass distribution. It sends the targeted DD a variety of valid and invalid I/O requests (not supported,

[1]The number of functions called from *NTOSKRNL.EXE*.
[2]The number of functions called from *HAL.DLL*.

malformed etc.) to reveal implementation vulnerabilities. The Table 7.1 lists the experimental outcomes and provides a comparative evaluation of the clustering strength (see Section 7.2.2).

Sandra was the workload that issued the highest number of distinct CSs (51 out of 9545), showing the highest cluster strength in the distinct CS population, with $AC = 0.859$. Also, the DD visited only three modes, intuitively indicating that this workload has the strongest tendency to reveal execution hotspots. At the other extreme, *Enable_Disable* only revealed 10 distinct CSs (out of 136), but instead the calls to the external functions were the most diverse, 38 from *NTOSKRNL.EXE* and 4 from *HAL.DLL*. As the agglomerative coefficient of this workload is relatively small, we expect that *Enable_Disable* has the weakest clustering tendency.

7.3.1 Revealing Execution Hotspots

To visualize the clustering tendency of the CSs belonging to the EPP as generated by the used workloads and, implicitly, the execution hotspots in floppy DD's code, we used a multidimensional scaling (MDS) plot. MDS is a statistical technique designed to graphically express the degree of similarity or dissimilarity between objects. The points representing similar objects are clustered together in different regions of the 2D-space depicted by the MDS plot, while the points representing dissimilar objects are placed far apart from each other. The MDS plot in Figure 7.6 is computed using the already available distance matrices.

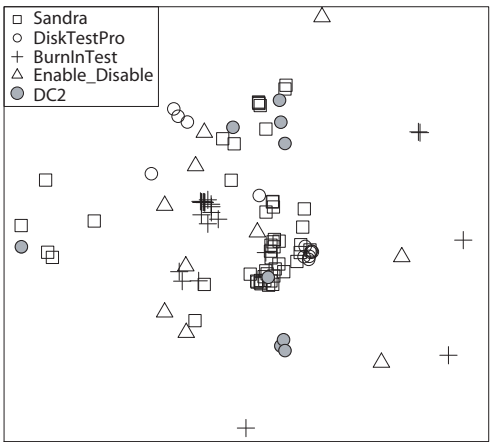

Figure 7.6: MDS plot of the CSs for each workload.

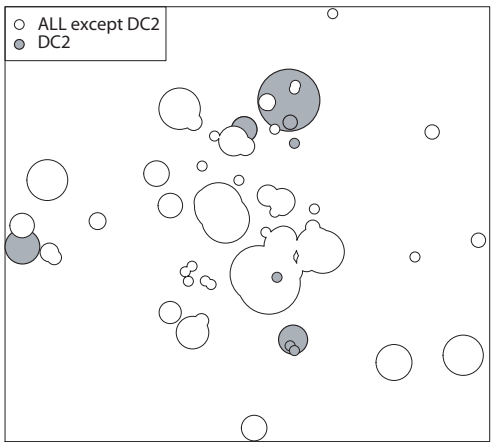

Figure 7.7: MDS plot of the execution hotspots with their magnitudes.

With a high AC, *Sandra* forms the biggest clusters mostly in the center of

the figure, while the areas exercised by the *Enable_Disable* are located farther apart from each other. This visual representation of the CSs also helped reveal another tight cluster close to the center of the Figure 7.6, generated by the *BurnInTest* workload. Also, *DiskTestPro*'s executions form a hotspot, located in the second quadrant of Figure 7.6. Overall, the grouping of the CSs in the middle of the MDS plot indicates that most of them share a certain degree of similarity (that is, most of the workloads executed the same code areas).

Interestingly, the EPP generated by *DC2* are located quite differently from the rest of other CSs. This is explained by the fact that *DC2* is a robustness testing tool, therefore accessing areas of code seldomly visited under common executions. To better substantiate this tendency, Figure 7.7 represents the same MDS plot, where each CS was enhanced with the magnitude of the associated CS. That is, a bigger circle represents a higher rate of occurrence of the respective CS.

The circles are scaled using a logarithmic function[2] in order to create a visual balance between CSs having different magnitudes. Additionally, the execution hotspots generated by the first four workloads from Table 7.1 were merged as white spots, while the hotspots generated by the robustness testing tool *DC2* were represented in gray. As *DC2*'s hotspots are off-centered, it becomes apparent that the *DC2* covers very few of the execution hotspots generated by all other studied workloads.

Nevertheless, figures 7.6 and 7.7 validate our methodology and graphically motivate the usage of execution profiles as a prerequisite step for testing. We believe that a significant amount of testing can be saved by redistributing the effort to covering the execution hotspots. Doing so significantly reduces the test effort, while the test adequacy remains unaffected. While test case filtering is not the scope of this thesis, we hypothesize that an iterative method based on comparisons of test suites against an existing execution hotspot map can be devised in order to guide this process.

7.3.2 Similarity Cutoffs

The dendrograms obtained at steps 5 and 6 in Figure 7.5 represent useful support for deciding which code paths to test. To ensure high accuracy for the subsequent testing campaigns with respect to the execution hotspots, the test cases must exercise the DD in the same manner as the workload does or, alternatively, use the test cases themselves as workload for exercising the DD in the profiling phase.

[2] $size = \log(magnitude_{CS})$

We believe that the testing effort can be significantly reduced by testing only the distinct CSs. A prioritization scheme for this procedure should consider (and therefore be indexed by) the occurrence index associated with each CS ($magnitude_{CS}$). Intuitively, a subsequent test campaign can reduce its overhead by testing only one CS per cluster.

Figure 7.8 illustrates this concept: by setting a similarity cutoff $T = 0.2$, the dendrogram is split into four clusters and five alones (CS0, 1, 15, 22 and 23). This indicates nine code paths that must be tested: the alones and any one CS from each of the four clusters, since all the CSs that are contained in the cluster are considered similar. With $T = 0$, 24 CSs should be tested in order to achieve complete hotspot coverage.

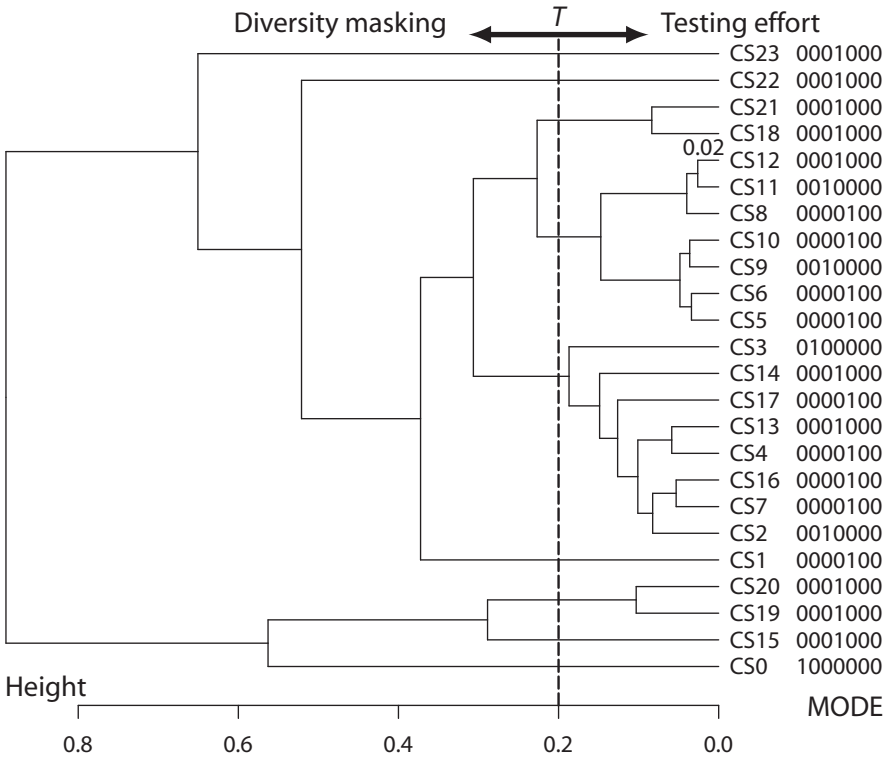

Figure 7.8: *BurnInTest*: A threshold set to 0.2 reveals a clustering with 4 clusters and 5 alones (62.5% test cost reduction).

Therefore, setting the $T = 0.2$ gives an overall reduction of 62.5% of the testing cost (assuming that the cost of testing is equally distributed among the 24 distinct CSs). In practice, the similarity cutoff T has to be chosen as close to zero as possible, because large values of T have a tendency to mask CS diversity. Actually, dendrograms support the similarity threshold

decision by their structure. If the CSs cluster at very low heights, a small cutoff value will group many CSs together, thus significantly reducing the test efforts without having to pay a high cost to diversity masking.

In contrast, Figure 7.9 depicts the dendrograms of the CSs for each mode. In this representation it is apparent that in the visited modes the DD was taking several different paths through the source code. The heights at which they cluster indicate that the CSs are quite dissimilar, even though they are basically associated with the same DD functionality. This reveals that the IRP dispatch routines are quite complex, possibly containing multiple decision branches in the code.

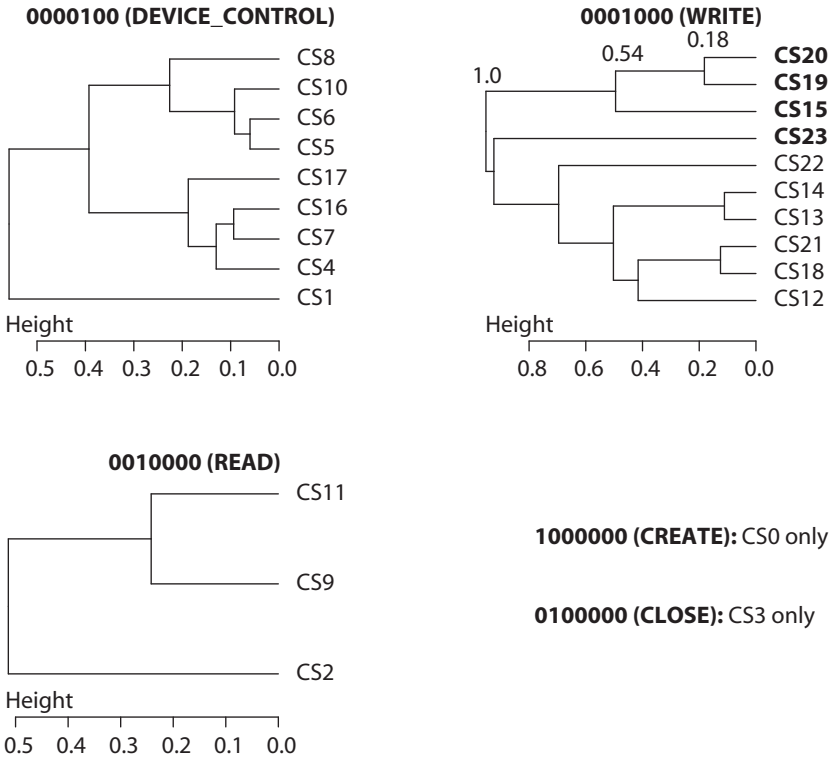

Figure 7.9: *BurnInTest*: The distinct CSs called by every mode.

In the case of the per-mode dendrograms (Figure 7.9), a similarity cutoff T smaller than the shortest cluster will reveal all the code paths taken inside the respective mode. Though, to balance the testing efforts, T should be chosen anywhere between the height of the smallest cluster and 1. With $T = 1$ the granularity of testing is the same as the one given by the DD's OP, namely the DD mode. Using the dual-interface approach presented in this chapter, a subsequent testing technique can take advantage of the smaller granularity offered by the new concept of *CS* versus the DD mode paradigm.

7.3.3 Results Interpretation

Sequence Types

From a testing perspective it is important to identify the code areas activated in the operational mode, thus permitting to accurately focus and prioritize subsequent testing activities. We distributed the CSs in equivalence classes based on their relative similarity (i.e., similar paths were grouped together) in order to emphasize the code areas exercised by a certain workload.

To reduce the testing effort without losing adequacy, we conjecture that only one CS per cluster needs to be tested. Our experimental results show that the code paths taken at runtime tend to differ significantly even if the DD is executing in the same DD mode. This is indicative of the high DD code complexity, warranting our effort toward providing testing assistance via methodical construction of the clusters, where the following are the key issues:

a. Similarity cutoff - the distance value among the intra-cluster objects (i.e., the degree of similarity among the objects of the same cluster). A small value results in many clusters containing fewer code paths (i.e., high testing overhead), while a too large value groups together code paths sharing little similarity, masking path diversity and thus reducing the adequacy of testing.

b. Similarity metric - the distance quantifier expressing the difference between any two code paths. It should properly capture and reward the features of the compared code paths accordingly. We have empirically used as similarity quantifier an equally-weighted average of Levenshtein and Jaro-Winkler similarity metrics [Cohen et al., 2003], as the emphasis was on presenting a valid methodology for DD profiling and not necessarily on its effectiveness for reducing testing effort.

To support an appropriate choice for the similarity metric, one of the research questions we plan to investigate is which code constructs generate the ascertained similarity classes. Currently, we distinguish three primary *similarity patterns* (SP) between any two captured DD code paths[3]:

SP1: [`xyabcz` and `mabcno`] - share a common substring (very often the same prefix); we believe that the same helper (or initialization) routine is performed by both runs;

SP2: [`xyabcz` and `mnabcabcabcabco`] - a common substring is repeated multiple times; we believe this is generated by a loop in the DD code;

[3]In the following, each character represents the encoding of a driver-external function called at runtime; for actual examples see Table 7.3.

SP3: [`abc` and `xyzmno`] - independent code paths; they should not be grouped in the same cluster as they need to be covered by different test cases.

Another research question that needs answering is how to decide which is the best-candidate code path for testing from each cluster, as the random choice might not always be acceptable. For instance, if a cluster contains four code paths out of which only three are very similar to each other, a random choice might elect the fourth one for testing, thus reducing the effectiveness of the equivalence partitioning as an abstraction for test reduction.

Identification of Repeating Functions

Table 7.3 depicts five distinct CSs, as generated by the *BurnInTest* workload. The respective CSs are highlighted also in Figure 7.9. CS15 is formed by a call `IofCompleteRequest` function, followed by `ExInterlockedRemoveHeadList` and `KeWaitForSingleObject`, repeating twice. The distance from CS15 to CS19 is 0.54 and to CS23 is 1.0; the distance from CS19 to CS20 is 0.18 (also depicted in Fig. 7.9). The low similarity values shared by the CS15, CS19 and CS20 are mainly given by the fact that the sequences share a common prefix and the group of two functions that repeat themselves. These repetitions indicate the presence of short loops in the DD's code.

In particular, according to the DDK documentation, `ExInterlockedRemoveHeadList` routine *"removes an entry from the head of a doubly linked list"* and `KeWaitForSingleObject` *"puts the current thread into a wait state"*. CS23 is heavily penalized when related to CS15 because the position of the only common character is not the same in the two CSs. In contrast, the distance from CS23 to CS0 is (only) 0.83 because both CSs are very short.

Table 7.2: Five functions and their encodings (used in Table 7.3).

Function Name	Encoding Char
`IofCompleteRequest`	a
`ExInterlockedRemoveHeadList`	b
`KeWaitForSingleObject`	c
`ExAcquireFastmutex`	d
`ExReleaseFastMutex`	e

Figure 7.8 shows cases when two CSs are very similar, even though they belong to different modes (i.e., CS11 and CS12, at a distance of 0.02). We believe that they share the same or large portions of a dispatch function. It is also possible that they share a large amount of helper functions. Future

Table 7.3: Four distinct CSs issued by the *BurnInTest*.

CS Name	Encoding	#Occurences
CS0	a	144
CS15	abcbc	2
CS19	abcbcbcbcbc	2
CS20	abcbcbcbc	1
CS23	dea	13

research directions include investigating in more depth the reasons behind this observed behavior on publicly available DD source code.

Table 7.4: The function calls accounting for 99.97% of all recorded calls, for all workloads, sorted descending on occurrence. The first two functions belong to *HAL.DLL*, the rest to *NTOSKRNL.EXE* library.

Rank	Function Name	#Occurrences	[%]
1	`ExAcquireFastMutex`	60414	18.40
2	`ExReleaseFastMutex`	60414	18.40
3	`IofCallDriver`	45976	14.00
4	`KeInitializeEvent`	40777	12.42
5	`IoBuildDeviceIoControlRequest`	40771	12.41
6	`ExInterlockedRemoveHeadList`	22007	6.70
7	`ExInterlockedInsertTailList`	16123	4.91
8	`IofCompleteRequest`	11562	3.52
9	`KeWaitForSingleObject`	11032	3.36
10	`KeReleaseSemaphore`	11003	3.35
11	`MmMapLockedPagesSpecifyCache`	8178	2.49
12	`MmPageEntireDriver`	24	0.01
13	`MmResetDriverPaging`	23	0.01
14	`KeGetCurrentThread`	10	0.00
15	`KeSetPriortyThread`	10	0.00
16	`ObfDereferenceObject`	10	0.00
17	`ObReferenceObjectByHandle`	10	0.00
18	`PsCreateSystemThread`	10	0.00
19	`PsTerminateSystemThread`	10	0.00
20	`ZwClose`	10	0.00

Frequently Used Kernel Services

Our profiling approach reveals that the set of functions frequently used by a DD at runtime is very small. Table 7.4 lists the 20 function calls that account for 99.97% of all the imports called by the *flpydisk.sys* at runtime in our experiments.

Mendonca and Neves chose a set of 20 DDK functions for fault injection experiments by inspecting the IAT tables of all the DDs belonging to several Windows installations [Mendonca and Neves, 2007]. In contrast, our results show that their static approach to select kernel APIs is irrelevant in such dynamic environments, as the set of functions called at runtime is radically different.

The targeted functions should be selected as a result of a profiling step similar to the profiling presented in this chapter in order to focus testing onto the services which are *actually* called by the DD in the operational phase. We believe that such a procedure saves testing resources while increasing the test adequacy and also the likelihood to find earlier the most relevant defects for the operational mode.

7.4 Chapter Summary

By simultaneously monitoring the *I/O call* and the *functional* interfaces of a selected DD we identified its *execution path profile* (EPP) as the set of distinct code paths taken by the DD, without requiring access to its source code.

As the captured code paths share a significant amount of similarity, they can be grouped in equivalence classes. The equivalence classes represent useful abstractions (execution hotspots) as they considerably simplify DD testing; if one code path from a cluster is tested, then all other code paths belonging to the same cluster are also considered tested. We also experimentally showed which are the parameters of primary importance in building clusters, namely the similarity metric and the clusterings's cutoff threshold.

The monitoring of the functional call interface of a DD revealed also another class of execution hotspots. This is represented by the set of intensely used kernel services implemented in OS libraries, therefore external to the monitored DD. The information (frequency, sequence, patterns, etc.) about the execution hotspots of the kernel libraries involved in the communication with the DDs might reveal fundamental defects in OS structures, as well as lead to performance or reliability enhancements.

Chapter 8

Conclusions and Future Research

What have we achieved in this thesis, and which are the future directions opened by the presented research efforts?

In this thesis we developed concepts and efficient methods for profiling the operational behavior of OS DDs. This chapter concludes it by summarizing the main contributions made, and discussing them from their applicability perspective. In this light, we speculate on the possible uses of our techniques in the DD development process for testing and debugging.

We believe that the work presented in this thesis opens up new interesting research directions. Therefore, this chapter discusses the key issues, and presents ideas for further enhancing the DD profiling methodology introduced in this thesis.

8.1 Overall Thesis Contributions

The main goal of the thesis was to develop mechanisms to permit an accurate assessment of the operational behavior of OS DDs. Our research effort was driven by the current need of profiling tools to improve the actual test methodologies for kernel-mode DDs. Accordingly, this section discusses the key contributions made by the research presented in this thesis. Driven by the research questions listed in Section 1.2.1 and grouped by topic, the thesis contributions are surveyed and their relevance is discussed.

8.1.1 Driver State Model and Test Space

As an effort to obtain the state space areas warranting focusing onto of the existing test tools, this thesis has developed a new state-based model describing a DD's operational evolution. The introduced *driver state model* is based only on the I/O request traffic available at the DD's interface with the OS kernel. The model is general enough to be representative for large classes of DDs, and it is developed for the most popular OS families, namely Microsoft Windows and UNIX/Linux. The driver state model in discussion was presented in depth in Chapter 3.

As an inherent development following the introduction of the driver model, in the same chapter we have defined the *total state space* that a DD can (theoretically) visit at runtime. Subsequently, Chapter 4 experimentally investigates which subset of the total state space is actually sojourned by existent Windows DDs. Termed *operational state space* (OSS), the visited subset of states represents only a small fraction of the total state space, as experimentally showed by the presented case studies on actual Windows DDs. The OSS therefore represents the area of the DD's state space which needs to primarily be covered by test tools in order to ensure adequacy and accuracy of the test outcomes. Moreover, the small size of the OSS hints that substantial test resources can be saved in contrast to the more traditional test methods.

Overall, the development of the driver state model and the highlighting of the operational state space for guiding subsequent test tools represent the contribution **C1** of our work, as defined in the introductory chapter of this thesis (Section 1.2.2).

Resultant publication

- **Constantin Sârbu**, Andréas Johansson, Falk Fraikin and Neeraj Suri, *Improving Robustness Testing of COTS OS Extensions*, in Proceed-

ings of the 3rd International Service Availability Symposium (ISAS), Helsinki, in Springer Verlag's LNCS 4328, pp. 120 – 139, 2006

8.1.2 Operational Profile

While the value of the OSS resides in its ability to discriminate between the visited and the non-visited DD states from the total state space, a more useful concept for testing is the *operational profile* (OP). The OP of a DD represents the set of DD states present in the OSS to whom occurrence and temporal quantifiers are assigned to. That is, each DD state is associated with a certain probability of occurrence in the field. In the OP the state transitions are also assigned with traversal probabilities. The state and transition probabilities are obtained via the operational mode quantifiers introduced in Chapter 5.

The same chapter contains in-depth case studies on several Windows XP and Vista DDs, illustrating the methodology used to obtain OPs for actual OS DDs. Actually, the body of our experimental work contains over fifty XP and Vista DDs, exercised using more then 260 workloads.

Chapter 6 describes several applications of the DD OPs for testing. Enabled by the operational quantifiers, the test prioritization via OP is investigated, followed by a comprehensive inter-workload comparison. The workload comparison allows for accurate estimations of the field workloads, alleviating the problem of finding realistic test workloads.

Next, several experimental issues are addressed, such as the overhead induced by our monitoring mechanisms, the effort required to obtain DD OPs and the threats to validity of our experimental procedures. Moreover, Chapter 6 discusses the significant test space reduction introduced by the OP versus the OSS of a given DD.

Concluding, the introduction of the operational mode quantifiers and the test prioritization methods using DD OPs, alongside with the extensive case studies presented in the chapters 5 and 6 represent the thesis contributions **C3**, **C4**, **C6**, **C7** and **C8**. For a detailed description of these contributions, see Section 1.2.2 of this thesis.

Resultant publications

- **Constantin Sârbu**, Andréas Johansson, Falk Fraikin and Neeraj Suri, *Improving Robustness Testing of COTS OS Extensions*, in Proceedings of the 3rd International Service Availability Symposium (ISAS), Helsinki, in Springer Verlag's LNCS 4328, pp. 120 – 139, 2006

- **Constantin Sârbu** and Neeraj Suri, *On Building (and Sojourning)*

the State-space of Windows Device Drivers, State-space Exploration for Automated Testing Workshop (SSEAT), Seattle, 2008

- **Constantin Sârbu**, Andréas Johansson, Neeraj Suri and Nachiappan Nagappan, *Profiling the Operational Behavior of OS Device Drivers*, in Proceedings of 19th International Symposium on Software Reliability Engineering (ISSRE), Seattle / Redmond, pp. 127 – 136, 2008

- **Constantin Sârbu**, Andréas Johansson, Neeraj Suri and Nachiappan Nagappan, *Profiling the Operational Behavior of OS Device Drivers*, submitted to the Empirical Software Engineering Journal, a special issue for the ISSRE 2008 best four papers (in review), 2009

8.1.3 Execution Path Profile

The OSS and the OP of a DD are built exclusively using the I/O request traffic between the respective DD and the OS kernel, without using source-code information of any of the involved OS kernel components. The central unit around which the concept of DD OP revolves is the driver state definition, termed as *driver mode*. The driver mode indicate the routines of the DD in execution at any time instant, but cannot precisely indicate where a defect is located if the DD crashes while executing in the respective mode. Therefore, for debugging purposes, we introduced in Chapter 7 the concepts of *call strings* (CS) and *execution path profile* (EPP).

The call strings are sequences of calls to functions implemented in kernel libraries which are external to the DD. In this thesis we consider the call strings as abstractions of the code paths taken at runtime by the given DD. The set of all CSs forms the execution path profile of the DD. The EPP contains many repeating CSs indicating common source code paths taken multiple times by the running DD in the operational phase.

Beside identical CS, our experiments showed that many CS share a high degree of similarity. Hence, using string similarity metrics, we grouped similar CSs into equivalence classes. Each equivalence class (or *cluster*) represents an execution hotspot, thus indicating possible performance and robustness bottlenecks warranting more attention from testers.

The introduced concepts of code paths and the execution path profiling methods presented in Chapter 7 are viable code profiling mechanisms, usable also in absence of source code. They represent the contributions **C2**, **C5** and **C8** of this thesis (see Section 1.2.2 for detailed descriptions thereof).

Resultant publications

- **Constantin Sârbu**, Andréas Johansson and Neeraj Suri, *Execution Path Profiling for OS Device Drivers: Viability and Methodology*, in Proceedings of the 5rd International Service Availability Symposium (ISAS), Tokyo, in Springer Verlag's LNCS 5017, pp. 90 – 109, 2008

- **Constantin Sârbu**, Nachiappan Nagappan and Neeraj Suri, *On Equivalence Partitioning of Code Paths inside OS Kernel Components*, in Proceedings of the First International Workshop on Software Technologies for Future Dependable Distributed Systems (STFSSD), Tokyo, 2009 (to appear)

8.2 Applications of Driver Profiling

Our metrics provide a useful quantification of the runtime behavior of a DD. As our OP quantifiers are statistical in nature, their relevance is directly proportional to the completeness of the workload used for obtaining them. Therefore, the choice of the workload used when building the OP is important and the monitoring phase should be long and diverse enough such that relevant behavior is captured. Such behavior is best captured if the monitoring is performed in the field, for instance during beta-testing or post-release monitoring. We have chosen commercial benchmarks to exercise our DDs as we believe they generate a mix of I/O requests with enough variety to be representative for the fashion DDs are used in the field.

Based on a non-intrusive state capture framework, our efforts provide accurate metrics and guidance for profiling and quantifying the runtime behavior for diverse classes of kernel-mode DDs. The presented experiments show the applicability of our approach into various phases of the DD development process, mainly for *testing* and *debugging*. Next, we speculate on the usability of our methods in this areas.

8.2.1 Testing

The empirical investigations detailed in this thesis show the applicability and the relevance of our state quantifiers for DD testing, as they:

- **reveal driver state sojourn patterns** without access to source code of the DDs;

- **assist test prioritization decision** based on quantified DD runtime profiles;

- **reduce the space size for testing activities** based on "tunable" coverage scenarios;

- **enable workload characterization and comparison** for selecting the most appropriate workload for in-house testing;

Subsequently, we detail the applicability of our DD profiling methodology as guidance for various aspects of the testing process. First, we show how the OP and EPP of a DD help reducing the overall cost of testing, by supporting in *test planning* (test process prioritization, test case generation), *test progress* (test coverage estimation) and *field data collection*.

Test Planning: Prioritization and Test Case Generation

From the test planning perspective, our approach provides a useful methodology to gain insight into the runtime behavior of the DD-under-test. It is not intended as a tool for finding bugs. Its main purpose is to quantify a DD's state sojourn behavior in order to guide testing towards certain subroutines or areas of code, but it does not directly reveal where the fault lies.

By prioritizing the test activities using the quantifiers proposed in this thesis, testers can increase the likelihood of finding sooner the "expensive" faults (i.e., the faults with higher probabilities to be activated post-release). As our methods enable identification of the execution hotspots in terms of both highly accessed DD routines (the OP) and in terms of often traversed DD code paths (the EPP), the subsequent test process can be "tuned" towards testing those areas first, or more thorough than other DD parts.

As the DD OP can be seen as a state machine representing the functionality of the DD for which it was captured, the OP can be used for automatic test case generation. In the formal methods research area, the generation of test cases using a state machine represents a hot research topic (for instance, see Hamon et al. [2004, 2005]). Therefore, we believe that once an accurate enough OP is available to describe the operational behavior of a DD, test cases for it can be automatically constructed, hence significantly reducing testing costs.

Another possibility to automatically obtain test cases via OP is the reuse of the captured I/O requests and the sequences thereof to build new test cases. Such I/O request sequences can be either "replayed" as-is on the DD as a test case, or its parameters can be altered in a fashion that matches the test purpose and then fed to the DD. Such an approach supposes that the relevant parameters of each I/O request are stored in the monitoring phase.

Test Progress Estimation

In many SW development projects testing is one of the last phases, directly preceding the actual release of the final SW product onto the market. Delays in other project phases often shorten the time and resources available for testing. This leads to unfortunate situations where the SW product is released before a minimal coverage level of testing can be achieved.

While the classical question "*When is the right moment to stop testing?*" still has only rule-of-thumb answers [Dalal and McIntosh, 1994; Huang and Lyu, 2005; Huang and Boehm, 2006; Musa and Ackerman, 1989], our OP model helps in quantitatively assessing at any moment in the testing phase how much of the test space was covered, and how many resources are still needed to reach the desired level of confidence. For instance, the cost of covering a DD mode belonging to the OP can be obtained from knowledge gained from the already tested modes. Using this, the duration and cost of covering the remaining of the test space can be accurately ascertained under the assumption that the test effort is equally distributed across the DD modes belonging to the OP.

Field Data Collection

Beside DD profiling, our state-aware DD monitoring method is relevant for field data collection as well. As many OS failures are currently caused by faulty DDs, adding state information to the collected crash reports can aid debugging by revealing the DD state history before a crash occurred. Currently, Microsoft collects crash reports as a part of *Watson* and *CEIP* projects [Orgovan, 2008].

By adding a wrapper around a DD targeted for field data collection (similarly to our filter driver), information about how the respective DD executes in user's setups can be gathered. This can be useful later on for various test purposes including failure and use-case data collection. Such wrappers can be placed as soon as the DD is in the beta-testing phase and even left after the final release, as the I/O and computational overheads induced by such a mechanism is relatively small (our filter driver's overhead was less than 3% – see Table 6.3).

8.2.2 Debugging

Beside helping in addressing test issues for DDs, our profiling methods enable also new insights for debugging. *Debugging* is the process in which a defect is first located and then fixed [Myers, 2004, chap. 7]. After debugging, parts

of the testing process have to be repeated (regression tests), as the big fixes might have introduced new faults.

If failure data captured in the field is available, it can be used to alleviate the process of locating the defect that produced the respective failure. If our OP is used, the process of locating the defect can be focused onto an I/O dispatch routine instead of larger source code parts. Instead, if the EPP of the DD is used, then the defect location can be identified in terms of the code path that produced the failure. As source-code level access to the DD is usually available for debugging, we believe that the insight provided by our profiling mechanisms enable fast and accurate defect localization.

8.3 Lessons Learned

The work presented in this thesis has developed a basis for profiling the runtime activity of DDs. First, we ascertained that despite of the low observability into the kernel, the operational activity of OS components can be successfully monitored. The prerequisite is that the monitoring mechanisms are themselves located in the kernel space, as user-space monitoring incurs a very high computational overhead, mainly given by the high frequency of context switching between user- and kernel-mode[1].

Second, DDs are located on the critical and inherently slow I/O path, thus compelling the monitoring components and any probes inserted into these paths to be *defect-free* and *fast*. For instance, in the layered driver architecture of Windows, an I/O request originating from an user-mode application must be passed by each driver to the next-lower one, until the hardware peripheral responsible for performing the actual I/O operation is reached. As they are acting within the kernel, the inserted probes must interact as little as possible with other OS structures, such that the probability of probe failures to propagate to critical OS entities is minimized.

Third, to ensure the validity of the obtained OPs and EPPs, the completeness of the captured communication flow versus has to be ensured. In the situation when the captured flow does not exactly match the actual flow (i.e., I/O requests are lost), the obtained DD profiles are useless. We learned that the critical issue in guaranteeing the logs completeness is the kernel buffer used to store the monitored traffic before it is saved to an actual log file. If the buffer size is small and the messages produced by the monitoring

[1]The *context switch* is an essential feature of a multitasking OS, enabling multiple tasks to share a single CPU resource. Transiting between user- and kernel-mode usually require context switches, during which the context of the running thread is saved and a new thread is started, after its state is loaded. Therefore, context switching is considered costly.

probe are issued at a very fast rate, the buffer might overflow, causing lost messages. As unbounded buffers are not allowed in kernel space, one possibility (used in our experimental work) is to keep the size of the monitoring messages as small as possible, to reduce the risk to overflow the buffer.

8.4 Open Ends - Basis for Future Work

While the work presented in this thesis addressed the research questions driving it towards making the discussed contributions, it also opened new and interesting research perspectives along its way. In the following, we briefly present some of the most promising ones.

Forcing the DD into a Specified Mode

One of the research questions we plan to further investigate is how to forcibly bring the DD into the modes of interest. As DDs perform in an arbitrary thread context and under the permanent influence of interrupts from the hardware devices, their runtime behavior is hard to predict. Especially for the modes where more than one dispatch routine is active this is not a trivial endeavor, as the DD might finish the processing for one I/O request before the other ones start. For instance, in Figure 6.1, the mode 010100 cannot be reached if the WRITE operation finishes before CLOSE is received. Hence, a simple test case that first calls WRITE followed immediately after by CLOSE might not necessarily bring the driver into the desired mode.

A possible solution might be to use the same workload that generated the driver's OP profile also for testing. As soon as a predecessor of the desired mode is reached, the parameters of the intercepted IRPs have to be changed on-the-fly and then fed to the DD *iff* the malformed IRP leads the DD into the mode of interest. This approach requires the development of mechanisms that detects the current state and keeps track of it at runtime. Such idea might work if the actual mechanisms for changing the IRP structure's parameters and state awareness are kept to a very low overhead in order not to disrupt the sequencing of I/O requests.

Profile-driven Test Tool

Using the lessons learnt from our DD monitoring approach, we are also considering to develop a technique for tuning an existing DD test tool to primarily cover the execution hotspots. The selection of test cases should consider the information obtained from a prior DD profiling phase in order to reduce the overall testing overhead. This will also help an early identification of the

insufficiently tested DD modes and assess their impact on DD robustness, together with an investigation of the possibility to correlate known OS failures to DD OPs.

Profile Detail Level

We intend to perform a detailed analysis of the presented state-based DD model in order to answer the question if increasing the detail richness of the collected data would actually help improve the accuracy of the model while keeping the abstraction level of a black-box.

Workload Choice

Another potential research direction is a quantitative study of driver-relevant workloads that considers using the profiling mechanisms introduced in this thesis to characterize workloads from a DD's perspective. This information would guide the choice of adequate workloads for specific test scenarios.

Code Paths vs. Program Control Graph

We also intend to map the obtained code paths to the control flow graphs of the DDs. This serves as validation of our black-box profiling methodology by quantifying its capacity to disclose the followed code paths. At the same time, we conjecture that this evaluative approach provides for a proper comparison of the available black- and white-box test methods for DDs from the code coverage perspective.

Bibliography

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986. ISBN 0-201-10088-6.

Arnaud Albinet, Jean Arlat, and Jean-Charles Fabre. Characterization of the impact of faulty drivers on the robustness of the Linux kernel. In *International Conference on Dependable Systems and Networks (DSN)*, pages 867–876, 2004. doi: 10.1109/DSN.2004.1311957.

Juan J. Amor, Gregorio Robles, Jesús M. González-Barahona, and Israel Herraiz. From Pigs to Stripes: A Travel through Debian. In *Proceedings of the Debian Annual Developers Meeting (DebConf5)*, 2005.

Jean Arlat, Alain Costes, Yves Crouzet, Jean-Claude Laprie, and David Powell. Fault Injection and Dependability Evaluation of Fault-Tolerant Systems. *IEEE Transaction on Computers*, 42(8):913–923, August 1993.

Jean Arlat, Jean-Charles Fabre, and Manuel Rodriguez. Dependability of COTS Microkernel-based Systems. *IEEE Transactions on Computers*, 51, issue d:138 – 163, 2002.

Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11 – 33, January 2004. doi: 10.1109/TDSC.2004.2.

Alberto Avritzer and Brian Larson. Load testing software using deterministic state testing. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 82 – 88, 1993. doi: 10.1145/154183.154244.

Thomas Ball and James R. Larus. Efficient path profiling. In *MICRO-29*, pages 46 – 57, December 1996. doi: 10.1109/MICRO.1996.566449.

Thomas Ball and Sriram Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *Symposium on Principles of Programming Languages*, pages 1 – 3, 2002.

Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. *SPIN Model Checking and Software Verification*, Springer Verlag's LNCS 1885:113 – 130, 2000. doi: 10.1007/10722468_7.

Thomas Ball and Sriram K. Rajamani. SLIC: A Specification Language for Interface Checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research, January 2001. URL `http://research.microsoft.com/pubs/69906/tr-2001-21.pdf`. Accessed on February 20th, 2009.

Thomas Ball, Byron Cook, Vladimir Levin, and Sriram Rajamani. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In *Integrated Formal Methods*, volume Springer-Verlag vol. 2999, pages 1–20, 2004.

Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, pages 73 – 85. ACM, 2006. ISBN 1-59593-322-0. doi: http://doi.acm.org/10.1145/1217935.1217943.

T. Bhat and N. Nagappan. Tempest: Towards early identification of failure-prone binaries. In *International Conference on Dependable Systems and Networks (DSN)*, pages 116–121, June 2008. doi: 10.1109/DSN.2008.4630079.

Barry W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

Shuo Chen, Jun Xu, Ravishankar K. Iyer, and Keith Whisnant. Evaluating the Security Threat of Firewall Data Corruption Caused by Instruction Transient Errors. In *International Conference on Dependable Systems and Networks (DSN)*, pages 495 – 504, 2002.

Ram Chillarege, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray, and Man-Yuen Wong. Orthogonal Defect Classification - A Concept for In-Process Measurements. *IEEE Transactions on Software Engineering (TSE)*, 18(11):943 – 956, November 1992.

Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson R. Engler. An empirical study of operating system errors. *ACM Symposium on Operating Systems Principles (SOSP)*, 1:73 – 88, 2001. URL `citeseer.ist.psu.edu/article/chou01empirical.html`.

Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 2001.

William W. Cohen, Ravikumar Pradeep, and Stephen E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *IJCAI*, pages 73 – 78, 2003. URL `citeseer.ist.psu.edu/cohen03comparison.html`.

Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly, 3 edition, February 2005. URL `http://www.oreilly.com/catalog/linuxdrive3/book/index.csp`.

S. R. Dalal and A. A. McIntosh. When to stop testing for large software systems with changing code. *IEEE Transactions on Software Engineering (TSE)*, 20(4):318 – 323, 1994. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.277579.

Edward N. Dekker and Joseph M. Newcomer. *Developing Windows NT Device Drivers: A Programmer's Handbook*. Addison-Wesley longman, Inc., 1999.

John DeVale and Philip Koopman. Performance Evaluation of Exception Handling in I/O Libraries. In *International Conference on Dependable Systems and Networks (DSN)*, pages 519 – 524, July 2001.

Devolo. MicroLink 56k Fun II, 2009. URL `http://www.devolo.com/co_EN/produkte/analog/ml56kfun2.html`. Accessed on March 6th, 2009.

William Dickinson, David Leon, and Andy Podgurski. Finding failures by cluster analysis of execution profiles. In *International Conference on Software Engineering, 2001 (ICSE)*, pages 339 – 348, 2001. doi: 10.1109/ICSE.2001.919107.

João Durães and Henrique Madeira. Characterization of operating systems behavior in the presence of faulty drivers through software fault emulation. In *Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 201 – 209, December 2002a. doi: 10.1109/PRDC.2002.1185639.

João Durães and Henrique Madeira. Emulation of software faults by educated mutations at machine-code level. *13th International Symposium on Software Reliability Engineering (ISSRE)*, pages 329 – 340, 2002b. ISSN 1071-9458. doi: 10.1109/ISSRE.2002.1173283.

João Durães and Henrique Madeira. Multidimensional characterization of the impact of faulty drivers on the operating systems behavior. *IEICE Transactions on Information and Systems*, 86(12):2563 – 2570, 2003. ISSN 09168532. URL `http://ci.nii.ac.jp/naid/110003213679/`.

João Durães and Henrique Madeira. Emulation of Software Faults: A Field Data Study and a Practical Approach. *IEEE Transactions on Software Engineering*, 32(11):849 – 867, 2006.

Eclipse Project. Eclipse Test and Performance Tools Platform Project, 2009. URL `http://www.eclipse.org/tptp/`. Accessed on February 19th, 2009.

N.E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering (TSE)*, 25, Issue: 5:675 – 689, Sept./Oct. 1999. doi: 10.1109/32.815326.

Christof Fetzer and Zhen Xiao. A Flexible Generator Architecture for Improving Software Dependability. In *International Symposium on Software Reliability Engineering (ISSRE)*, pages 102 – 113, 2002a.

Christof Fetzer and Zhen Xiao. An Automated Approach to Increasing the Robustness of C Libraries. In *International Conference on Dependable Systems and Networks (DSN)*, pages 155 – 164, June 2002b.

Archana Ganapathi and David Patterson. Crash Data Collection: a Windows Case Study. In *International Conference on Dependable Systems and Networks (DSN)*, pages 280 – 285, 2005. doi: 10.1109/DSN.2005.32.

Archana Ganapathi, Viji Ganapathi, and David Patterson. Windows XP Kernel Crash Analysis. In *Large Installation System Administration Conference (LISA)*, pages 12 – 22, 2006.

Jim Gray. Why Do Computers Stop and What Can We Do About It. Technical Report TR 85.5, Tandem, 1985.

Jim Gray. A Census of Tandem System Availability Between 1985 and 1990. *IEEE Transactions on Reliability*, 39(4):409 – 418, 1990. ISSN 0018-9529.

Richard W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 26(2):147 – 160, 1950.

Grégoire Hamon, Leonardo deMoura, and John Rushby. Generating efficient test sets with a model checker. *2nd International Conference on Software Engineering and Formal Methods*, pages 261 – 270, 2004.

Grégoire Hamon, Leonardo deMoura, and John Rushby. Automated Test Generation with SAL. Technical report, CSL Technical Note, January 2005.

Ahmed E. Hassan, Daryl J. Martin, Parminder Flora, Paul Mansfield, and Dave Dietz. An industrial case study of customizing operational profiles using log compression. In *30th international Conference on Software Engineering (ICSE)*, pages 713 – 723. ACM, 2008. ISBN 978-1-60558-079-1. doi: http://doi.acm.org/10.1145/1368088.1379445.

Jane Huffman Hayes and Jeff Offutt. Input validation analysis and testing. *Empirical Software Engineering (EMSE)*, 11(4):493 – 522, December 2006.

Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Failure Resilience for Device Drivers. In *International Conference on Dependable Systems and Networks (DSN)*, pages 41 – 50, 2007. doi: 10.1109/DSN.2007.46.

Martin Hiller, Arshad Jhumka, and Neeraj Suri. PROPANE: An Environment for Examining the Propagation of Errors in Software. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 81 – 85, July 2002.

Martin Hiller, Arshad Jhumka, and Neeraj Suri. EPIC: Profiling the Propagation and Effect of Data Errors in Software. *IEEE Transactions on Computers*, 53(5):512 – 530, May 2004.

Jason Hiner. Sanity check: Five reasons why Windows Vista failed, October 2008. URL `http://blogs.techrepublic.com.com/hiner/?p=849&tag=nl.e055`. Accessed on February 5th, 2009.

Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault Injection Techniques and Tools. *IEEE Computer*, 30(4):75 – 82, April 1997.

Chin-Yu Huang and M.R. Lyu. Optimal release time for software systems considering cost, testing-effort, and test efficiency. *IEEE Transactions on Reliability*, 54(4):583 – 591, December 2005. ISSN 0018-9529. doi: 10.1109/TR.2005.859230.

L. Huang and B. Boehm. How much software quality investment is enough: A value-based approach. *IEEE Software*, 23(5):88 – 95, Sept.-Oct. 2006. ISSN 0740-7459. doi: 10.1109/MS.2006.127.

Galen Hunt and Doug Brubacher. Detours: Binary Interception of Win32 Functions. In *3rd USENIX Windows NT Symposium*, pages 135 – 144, July 1999.

IEEE. IEEE Standard Glossary of Software Engineering. IEEE Standard 610.12-1990, December 1990.

Ross Ihaka and Robert Gentleman. R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics*, 5(3):299 – 314, 1996.

Matthew A. Jaro. Advances in Record-Linkage Methodology as Applied to Matching the 1985 Census of Tampa, Florida. *Journal of the American Statistical Association*, 84(406):414 – 420, 1989. ISSN 01621459. URL `http://www.jstor.org/stable/2289924`.

Steven P. Jobs. Keynote talk at Apple World Wide Developers Conference (WWDC), 2006.

Andréas Johansson and Neeraj Suri. Error propagation profiling of operating systems. In *International Conference on Dependable Systems and Networks (DSN)*, pages 86– – 95, 2005. doi: 10.1109/DSN.2005.45.

Andréas Johansson, Adina Sârbu, Arshad Jhumka, and Neeraj Suri. On enhancing the robustness of commercial operating systems. *International Service Availability Symposium (ISAS)*, Springer LNCS 3335:148 – 159, May 2004.

Andréas Johansson, Neeraj Suri, and Brendan Murphy. On the Impact of Injection Triggers for OS Robustness Evaluation. In *International Symposium on Software Reliability Engineering (ISSRE)*, pages 127 – 136, 2007a.

Andréas Johansson, Neeraj Suri, and Brendan Murphy. On the Selection of Error Model(s) for OS Robustness Evaluation. In *International Conference on Dependable Systems and Networks (DSN)*, pages 502 – 511, 2007b.

S. C. Johnson and S. C. Johnson. Lint, a C Program Checker. In *Computer Science Technical Reports*, pages 78 – 1273. Murray Hill, 1978.

Cem Kaner, Jack Falk, and Hung Q. Nguyen. *Testing Computer Software*. John Wiley & Sons, 1999.

Wei-Lun Kao, Ravishankar K. Iyer, and Dong Tang. FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under

Faults. *IEEE Transactions on Software Engineering (TSE)*, 19(11):1105 – 1118, November 1993.

Rob Knies. Providing a Template for Tech Transfer, October 2005. URL `http://research.microsoft.com/en-us/news/features/slam.aspx`. Accessed online Feb. 20th, 2009.

Philip Koopman and John DeVale. Comparing the Robustness of POSIX Operating Systems. In *International Symposium on Fault-Tolerant Computing*, pages 72 – 79, 1999.

Philip Koopman and John DeVale. The Exception Handling Effectiveness of POSIX Operating System. *IEEE Transactions on Software Engineering (TSE)*, 26(9):837 – 848, September 2000.

Jean-Claude Laprie, editor. *Dependability: Basic Concepts and Terminology.* Springer Verlag, 1992.

James R. Larus. Whole program paths. In *ACM SIGPLAN*, volume 34, pages 259 – 269, May 1999. doi: 10.1145/301631.301678.

Lucas Layman, Gunnar Kudrjavets, and Nachiappan Nagappan. Iterative identification of fault-prone binaries using in-process metrics. In *ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 206 – 212. ACM, 2008. ISBN 978-1-59593-971-5. doi: http://doi.acm.org/10.1145/1414004.1414038.

David Leon and Andy Podgurski. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In *14th International Symposium on Software Reliability Engineering (ISSRE)*, pages 442 – 453, 2003. doi: 10.1109/ISSRE.2003.1251065.

V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. Technical Report 8, Soviet Physics Doklady, 1966.

Vincent Maraia. How Many Lines of Code in Windows?, December 2005. URL `http://www.knowing.net/index.php/2005/12/06/how-many-lines-of-code-in-windows/`. Accessed online Feb. 3rd, 2009.

Scott McMaster and Atif M. Memon. Call stack coverage for test suite reduction. In *IEEE International Conference on Software Maintenance (ICSM)*, pages 539 – 548, 2005. doi: 10.1109/ICSM.2005.29.

Manuel Mendonca and Nuno Neves. Robustness Testing of the Windows DDK. In *International Conference on Dependable Systems and Networks (DSN)*, pages 554 – 564, June 2007. doi: 10.1109/DSN.2007.85.

Microsoft. Windows Roadmap for Drivers, October 2006. URL `http://www.microsoft.com/whdc/driver/foundation/DrvRoadmap.mspx`.

Microsoft Corp. Windows NT Hardware Abstraction Layer (HAL). Microsoft Help and Support, October 2006. URL `http://support.microsoft.com/kb/99588`. Accessed on February 26th, 2009.

Microsoft Corp. Visual Studio, Microsoft Portable Executable and Common Object File Format Specification. Technical report, Microsoft Corporation, March 2008. URL `http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx`. Accessed on March 17th, 2009.

Microsoft Corp. Driver Verifier, 2009a. URL `http://www.microsoft.com/whdc/DevTools/tools/DrvVerifier.mspx`. Accessed on January 6th. 2009.

Microsoft Corp. Windows Logo Kit, 2009b. URL `http://www.microsoft.com/whdc/winlogo/WLK/default.mspx`. Accessed on March 6th, 2009.

Microsoft Corp. Windows Driver Kit for Windows 7 Beta: Announcements, February 5th 2009c. URL `http://www.microsoft.com/whdc/DevTools/WDK/WDKbeta.mspx`. Accessed on February 26th, 2009.

Microsoft Corp. Windows Hardware Developer Central – Static Driver Verifier, 2009d. URL `http://www.microsoft.com/whdc/devtools/tools/sdv.mspx`. Accessed on February 20th, 2009.

K.-H. Möller and D.J. Paulish. An empirical investigation of software fault distribution. In *First International Software Metrics Symposium*, pages 82 – 90, May 1993. doi: 10.1109/METRIC.1993.263798.

Brendan Murphy. Automating Software Failure Reporting. *Queue*, 2(8):42 – 48, 2004.

Brendan Murphy and Björn Levidow. Windows 2000 Dependability. In *Workshop on Dependable Networks and OS*, 2000.

Brendan Murphy, Mario Garzia, and Neeraj Suri. Closing the gap in failure analysis. In *DSN*, pages 59 – 61, 2006.

J.D. Musa and A.F. Ackerman. Quantifying software validation: When to stop testing? *IEEE Software*, 6(3):19 – 27, May 1989. ISSN 0740-7459. doi: 10.1109/52.28120.

John D. Musa. The operational profile in software reliability engineering: An overview. *International Symposium on Software Reliability Engineering (ISSRE)*, pages 140 – 154, October 1992. doi: 10.1109/ISSRE.1992.285850.

John D. Musa. Operational profiles in software reliability engineering. *IEEE Software*, 10(2):14 – 32, March 1993. ISSN 0740-7459. doi: 10.1109/52.199724.

John D. Musa. Sensitivity of field failure intensity to operational profile errors. In *International Symposium on Software Reliability Engineering (ISSRE)*, pages 334 – 337, November 1994a. doi: 10.1109/ISSRE.1994.341399.

John D. Musa. Adjusting measured field failure intensity for operational profile variation. In *International Symposium on Software Reliability Engineering (ISSRE)*, pages 330 – 333, November 1994b. doi: 10.1109/ISSRE.1994.341398.

John D. Musa. Software reliability-engineered testing. *Computer*, 29(11):61 – 68, Nov 1996. ISSN 0018-9162. doi: 10.1109/2.544239.

John D. Musa. *Software Reliability Engineering: More Reliable Software Faster and Cheaper*. AuthorHouse, 2nd edition, 2004.

John D. Musa and K. Okumoto. A Logarithmic Poisson Execution Time Model for Software Reliability Measurement. In *International Conference on Software Engineering (ICSE)*, pages 230 – 238, 1984.

Glenford J. Myers. *The Art of Software Testing*. Wiley & Sons, inc., 2nd edition, 2004.

Nacchiappan Nagappan, Laurie Williams, Jason Osborne, Mladen Vouk, and Pekka Abrahamsson. Providing test quality feedback using static source code and automatic test suite metrics. In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 83 – 94, 2005. doi: 10.1109/ISSRE.2005.35.

Nachiappan Nagappan, Brendan Murphy, and Victor Basili. The influence of organizational structure on software quality: An empirical case study. In *International Conference on Software Engineering (ICSE)*, pages 521

– 530. ACM, 2008. ISBN 978-1-60558-079-1. doi: http://doi.acm.org/10.1145/1368088.1368160.

S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443 – 453, March 1970. URL `http://view.ncbi.nlm.nih.gov/pubmed/5420325`.

Nuno Ferreira Neves, João Antunes, Miguel Correia, Paulo Veríssimo, and Rui Neves. Using attack injection to discover new vulnerabilities. In *International Conference on Dependable Systems and Networks (DSN)*, pages 457 – 466, 2006.

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Verlag, 2005.

Walter Oney. *Programming the MS Windows Driver Model*. Microsoft Press, Redmond, Washington, 2003.

Open Systems Resources, Inc. IrpTracker, 2009. URL `http://www.osronline.com/article.cfm?article=199`. Accessed on March 6th, 2009.

Vincent Orgovan. Windows Feedback and Reliability. Keynote talk at the 19th International Symposium on Software Reliability Engineering (ISSRE), November 2008. URL `http://www.csc2.ncsu.edu/conferences/issre/2008/VinceO_ISSRE_2008.pdf`. Accessed on Feburary 5th, 2009.

Penny Orwick and Guy Smith. *Developing Drivers with the Windows Driver Foundation*. Microsoft Press, Redmond, Washington, 2007.

Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Where the bugs are. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 86 – 96. ACM, 2004. ISBN 1-58113-820-2. doi: http://doi.acm.org/10.1145/1007512.1007524.

Jiantao Pan, Philip Koopman, Daniel Siewiorek, Yennun Huang, Robert Gruber, and Mimi Ling Jiang. Robustness Testing and Hardening of CORBA ORB Implementations. In *International Conference on Dependable Systems and Networks (DSN)*, pages 141 – 150, 2001.

Passmark Software. Modem Test, 2007. URL `http://www.passmark.com/products/modemtst.htm`. Accessed on March 6th, 2009.

Rational Inc. Purify, 2009. URL `http://www-01.ibm.com/software/awdtools/purify`. Accessed on January 9th, 2009.

G. Rothermel, R.H. Untch, Chengyun Chu, and M.J.; Harrold. Prioritizing test cases for regression testing. *Transactions on Software Engineering (TSE)*, 26:929 – 948, October 2001. doi: 10.1109/32.962562.

Mark Russinovich. DebugView, 2008. URL `http://technet.microsoft.com/en-us/sysinternals/bb896647.aspx`.

Charles P. Shelton, Philip Koopman, and Kobey DeVale. Robustness Testing of the Microsoft Win32 API. In *International Conference on Dependable Systems and Networks (DSN)*, 2000.

Abraham Silberschatz, Peter Baer Galvin, and Greg Gagnet. *Operating Systems Concepts*. John Wiley & Sons, 7th edition, dECEMBER 2004.

Michele Simionato. An Introduction to GraphViz and dot, 2004. URL `http://www.linuxdevcenter.com/pub/a/linux/2004/05/06/graphviz_dot.html`.

Daniel Simpson. Windows XP Embedded with Service Pack 1 Reliability. Technical report, Microsoft Corporation, 2003. URL `http://msdn2.microsoft.com/en-us/library/ms838661.aspx`. Accessed on October 10th, 2007.

SLAM2.0. SLAM, 2009. URL `http://research.microsoft.com/en-us/projects/slam/`. Accessed on February 20th, 2009.

T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195 – 197, 1981. ISSN 0022-2836. doi: DOI:10.1016/0022-2836(81)90087-5. URL `http://www.sciencedirect.com/science/article/B6WK7-4DN3Y5S-24/2/b00036bf942b543981e4b5b7943b3f9a`.

Constantin Sârbu. Additional Operational Profiles, 2009. URL `http://www.deeds.informatik.tu-darmstadt.de/research/OPs`. Accessed online on March 25th, 2009.

Mark Sullivan and Ram Chillarege. Software defects and their impact on system availability - a study of field failures in operating systems. In *International Symposium Fault-Tolerant Computing*, pages 2 – 9, 1991.

Mark Sullivan and Ram Chillarege. A comparison of software defects in database management systems and operating systems. In *International Symposium on Fault-Tolerant Computing*, pages 475 – 484, 1992.

Sun Microsystems. Java Virtual Machine Profiler Interface, 2009a. URL `http://java.sun.com/j2se/1.5.0/docs/guide/jvmpi/jvmpi.html`.

Sun Microsystems. Java Virtual Machine Tool Interface, 2009b. URL `http://java.sun.com/j2se/1.5.0/docs/guide/jvmpi/jvmpi.html`.

Martin Süßkraut and Christof Fetzer. Automatically Finding and Patching Bad Error Handling. In *European Dependable Computing Conference (EDCC)*, pages 13 – 22, 2006.

Martin Süßkraut and Christof Fetzer. Robustness and Security Hardening of COTS Software Libraries. In *International Conference on Dependable Systems and Networks (DSN)*, pages 61 – 71, 2007.

Michael M. Swift, Steven Martin, Henry M. Levy, and Susan J. Eggers. Nooks: an Architecture for Reliable Device Drivers. In *Workshop on ACM SIGOPS European Workshop (EW10)*, pages 102 – 107. ACM, 2002. doi: http://doi.acm.org/10.1145/1133373.1133393.

Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 23(1):77 – 110, 2005. ISSN 0734-2071. doi: 10.1145/1047915.1047919.

Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2 edition, 2001.

Amit Vasudevan and Ramesh Yerraballi. SPiKE: Engineering Malware Analysis Tools using Unobtrusive Binary-Instrumentation. In *Australasian Computer Science Conference (ACSC)*, pages 311 – 320, 2006.

E. J. Weyuker and A. Avritzer. A metric for predicting the performance of an application under a growing workload. *IBM Systems Journal*, 41(1):45 – 54, 2002. doi: 10.1147/sj.411.0045.

E.J. Weyuker. Testing component-based software: A cautionary tale. *IEEE Software*, 15 , Issue: 5:54 – 59, 1998. doi: 10.1109/52.714817.

Elaine J. Weyuker. Using operational distributions to judge testing progress. In *ACM Symposium on Applied Computing*, pages 1118 – 1122. ACM Press, 2003. ISBN 1-58113-624-2. doi: 10.1145/952532.952750.

Elaine J. Weyuker and Bingchiang Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering (TSE)*, 17, Issue: 7:703 – 711, July 1991. doi: 10.1109/32.83906.

David A. Wheeler. Estimating Linux's Size, 2000. URL `http://www.dwheeler.com/sloc`. Accessed on February 3rd, 2009.

David A. Wheeler. More Than a Gigabuck: Estimating GNU / Linux's Size, 2001. URL `http://www.dwheeler.com/sloc`. Accessed on December, 12th, 2008.

James A. Whittaker. What is software testing? and why is it so hard? *IEEE Software*, 17(1):70 – 79, 2000. ISSN 0740-7459. doi: http://dx.doi.org/10.1109/52.819971.

James A. Whittaker. *How to Break Software*. Addison-Wesley, 2003.

William E. Winkler. The state of record linkage and current research problems. Technical report, Statistical Research Division, U.S. Census Bureau, 1999.

Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Networked Windows NT System Field Failure Data Analysis. In *Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 178 – 185, 1999.

Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. SafeDrive: Safe and Recoverable Extensions Using Language-based Techniques. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 45 – 60. USENIX Association, 2006. ISBN 1-931971-47-1.

Index

Curriculum Vitae

Personal Data

Name: Constantin Sârbu

Date of birth: July 6th, 1976

Place of birth: Bucharest, Romania

School Education

1983-1991 Şcoala Generală nr.171 – "P. Ispirescu", Bucharest, Romania

1991-1995 Informatics Department, "Aurel Vlaicu" High School, Bucharest, Romania

University Education

1995-2000 *Dipl. Engineer in Automatic Control and Computer Science* – Faculty of Automatic Control and Computer Science, Polytechnic University of Bucharest, Romania

2000-2001 *Advanced Studies in Databases and Open Systems* – Faculty of Automatic Control and Computer Science, Polytechnic University of Bucharest, Romania

2002-2003 *Research Assistant* – Max Planck Institute for Brain Research, Frankfurt am Main, Germany

2003-2009 *Ph.D. in Computer Science* – Technische Universität Darmstadt, Darmstadt, Germany